

---

# Testing en Java

Tests unitarios con TDD

---

# Introducción a los tests

- Beneficios de los tests
- Tests automáticos vs manuales
- Tipos de tests



---

# Beneficios de los tests

- Comprobar los requerimientos
- Documentación / ejemplos
- Ayudar en el diseño (TDD)
- Confianza al desarrollar
- Confianza para refactorizar (evitando deuda técnica)
- ¡Herramienta que te da valor!

---

# Tests automáticos (vs tests manuales)

- + Automáticos
- + Rápidos
- + Fiables
- + Incrementales
- Tiempo de desarrollo
- No siempre viables



---

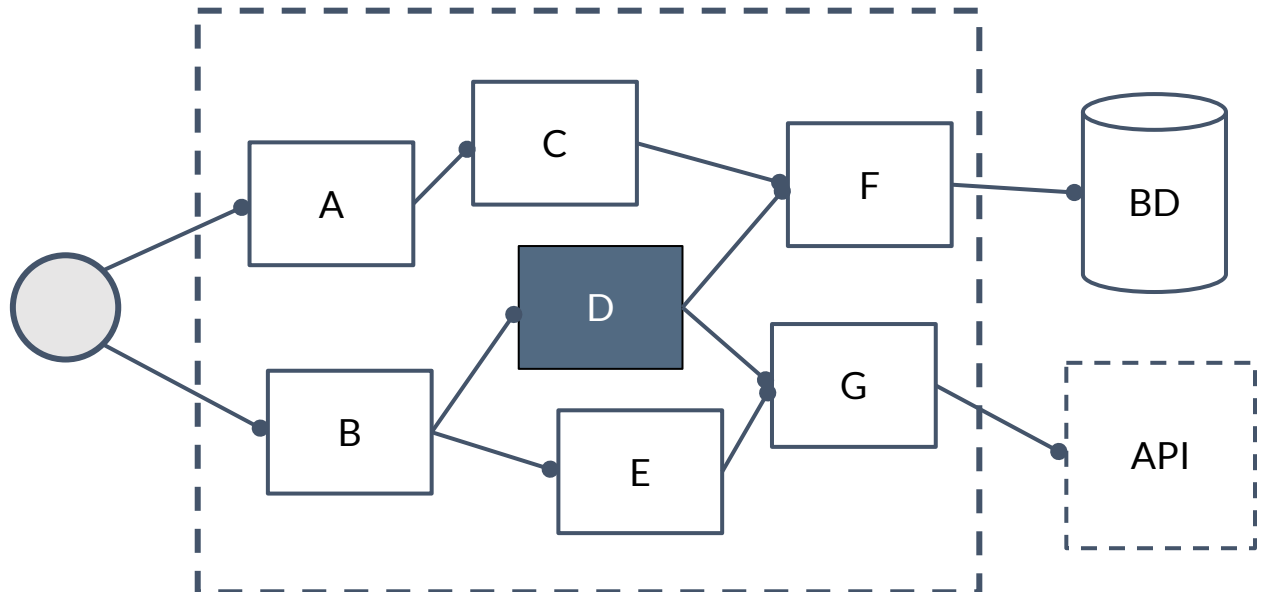
# Tipos de tests

- Unitarios
- Integración
- Funcionales
- Inicio a fin (end to end)
- Estrés

---

# Tipos de tests

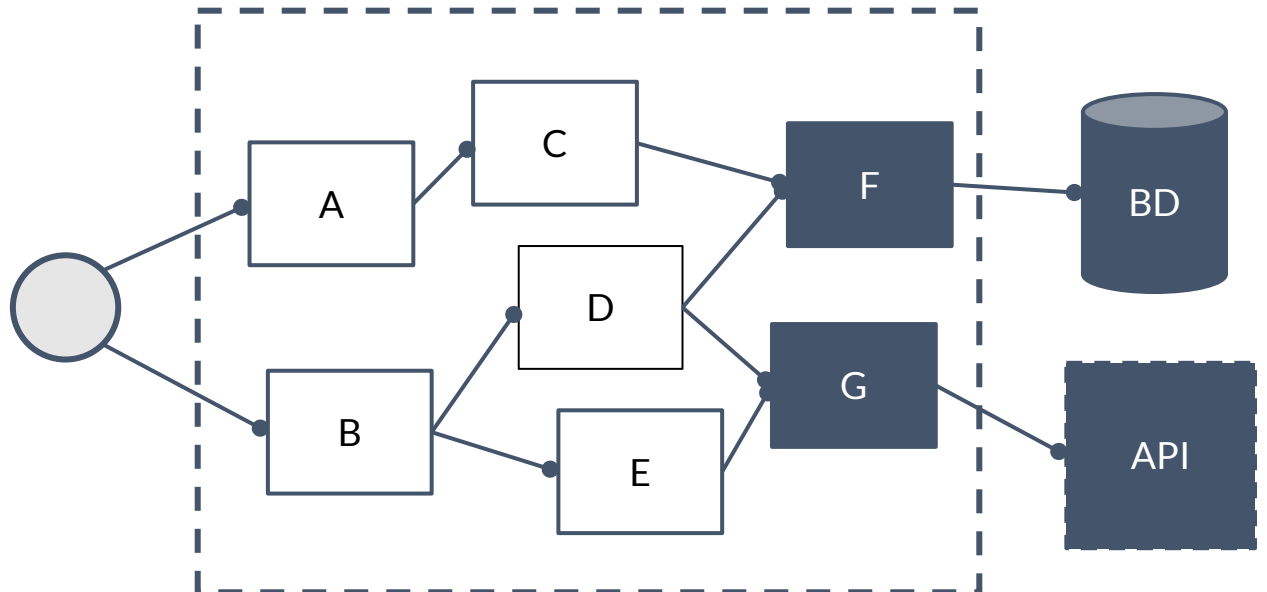
- Unitarios
- Integración
- Funcionales
- Inicio a fin (end to end)
- Estrés



---

# Tipos de tests

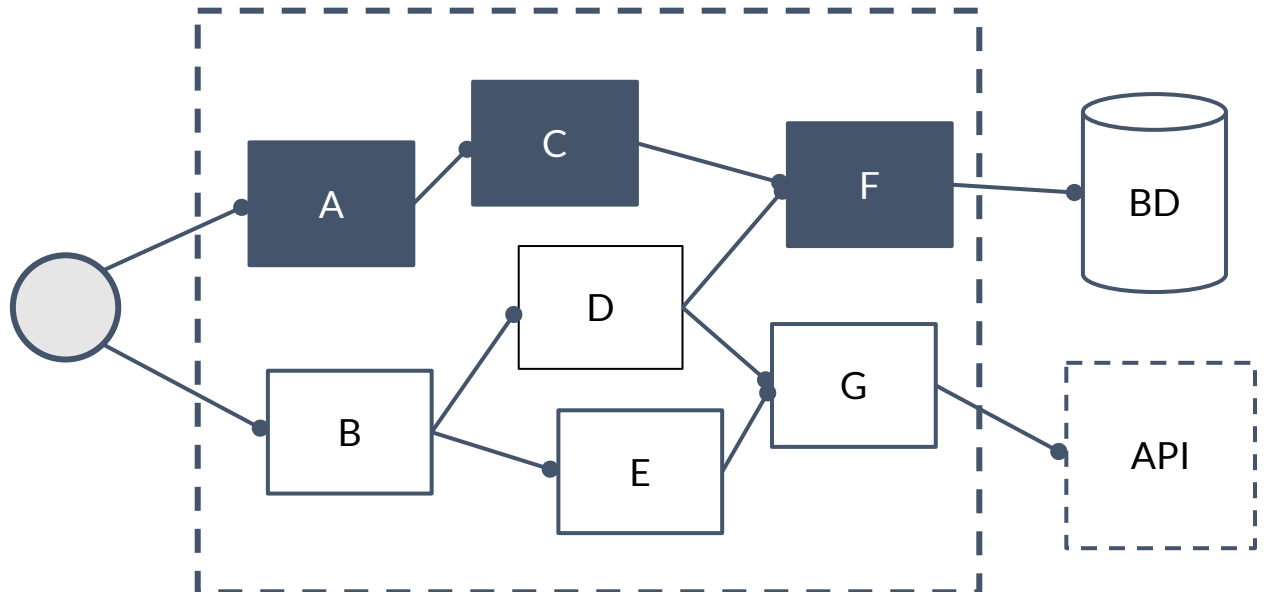
- Unitarios
- Integración
- Funcionales
- Inicio a fin (end to end)
- Estrés



---

# Tipos de tests

- Unitarios
- Integración
- **Funcionales**
- Inicio a fin (end to end)
- Estrés

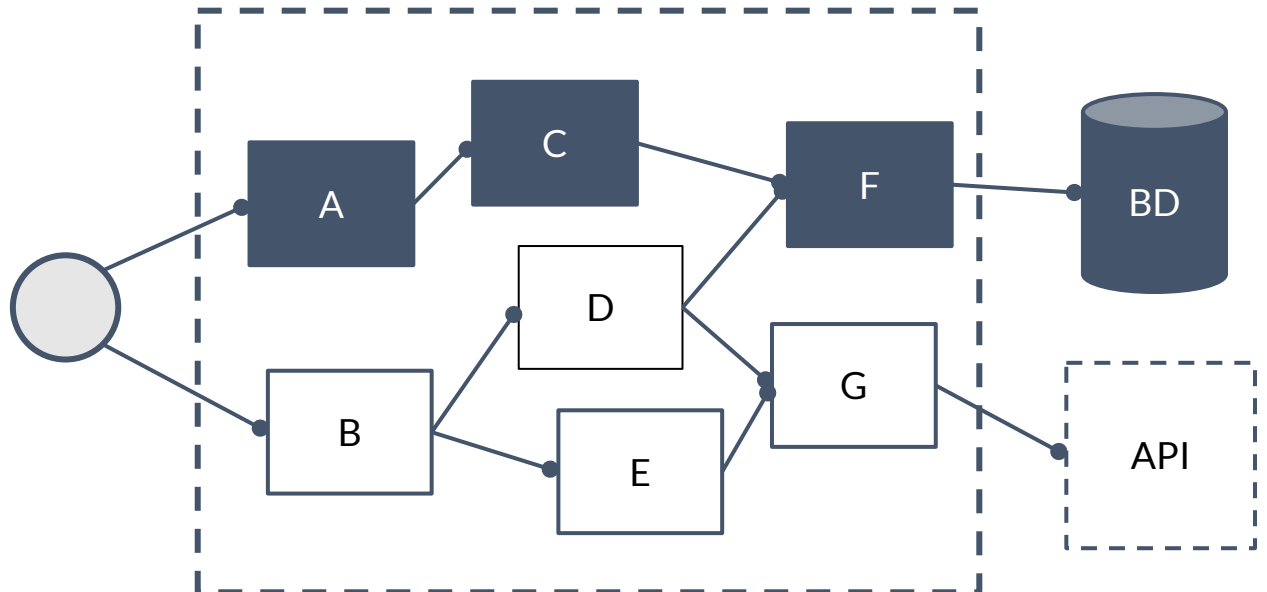




---

# Tipos de tests

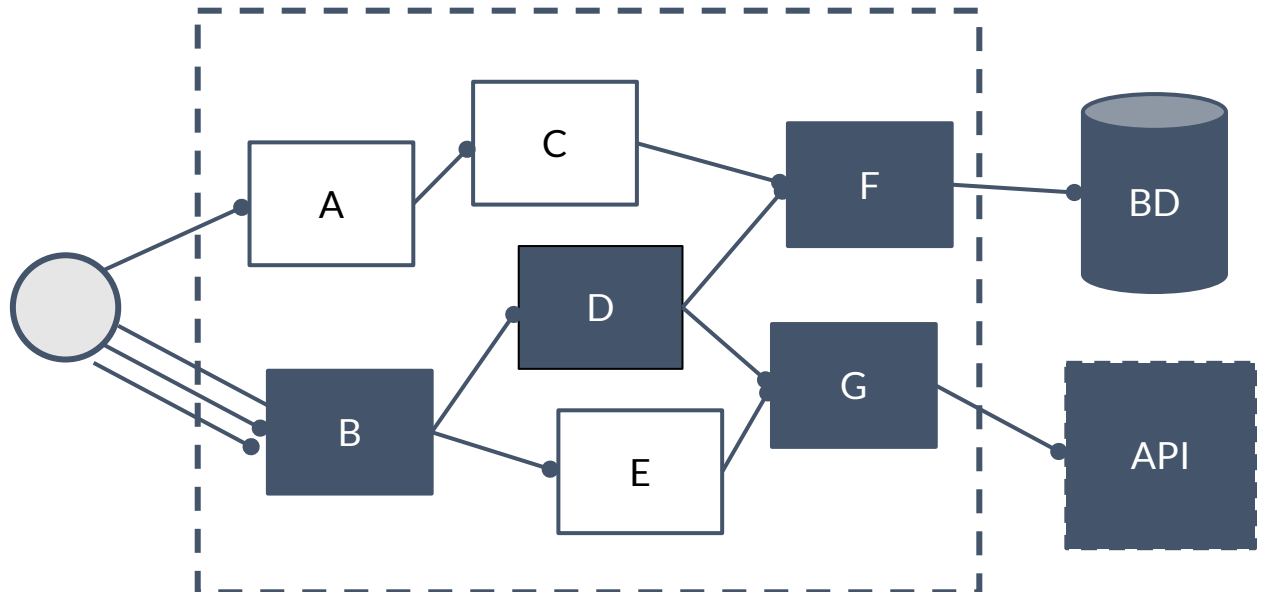
- Unitarios
- Integración
- Funcionales
- **Inicio a fin (end to end)**
- Estrés



---

# Tipos de tests

- Unitarios
- Integración
- Funcionales
- Inicio a fin (end to end)
- **Estrés**



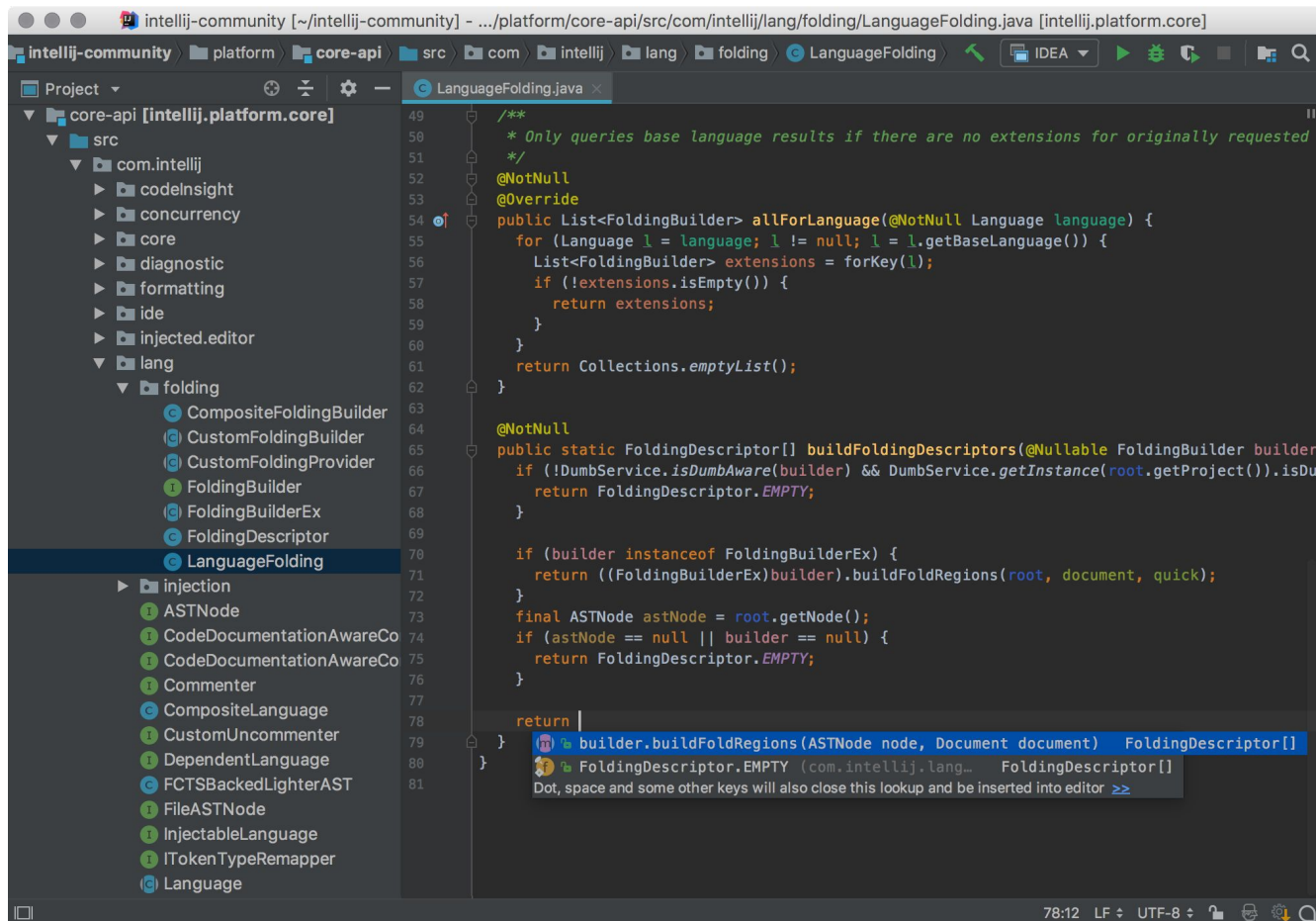
---

# Preparación del IDE, proyecto y librerías

- IDE: IntelliJ IDEA
- Proyecto Maven
- Test automático
- Test con JUnit
- JUnit + Mockito

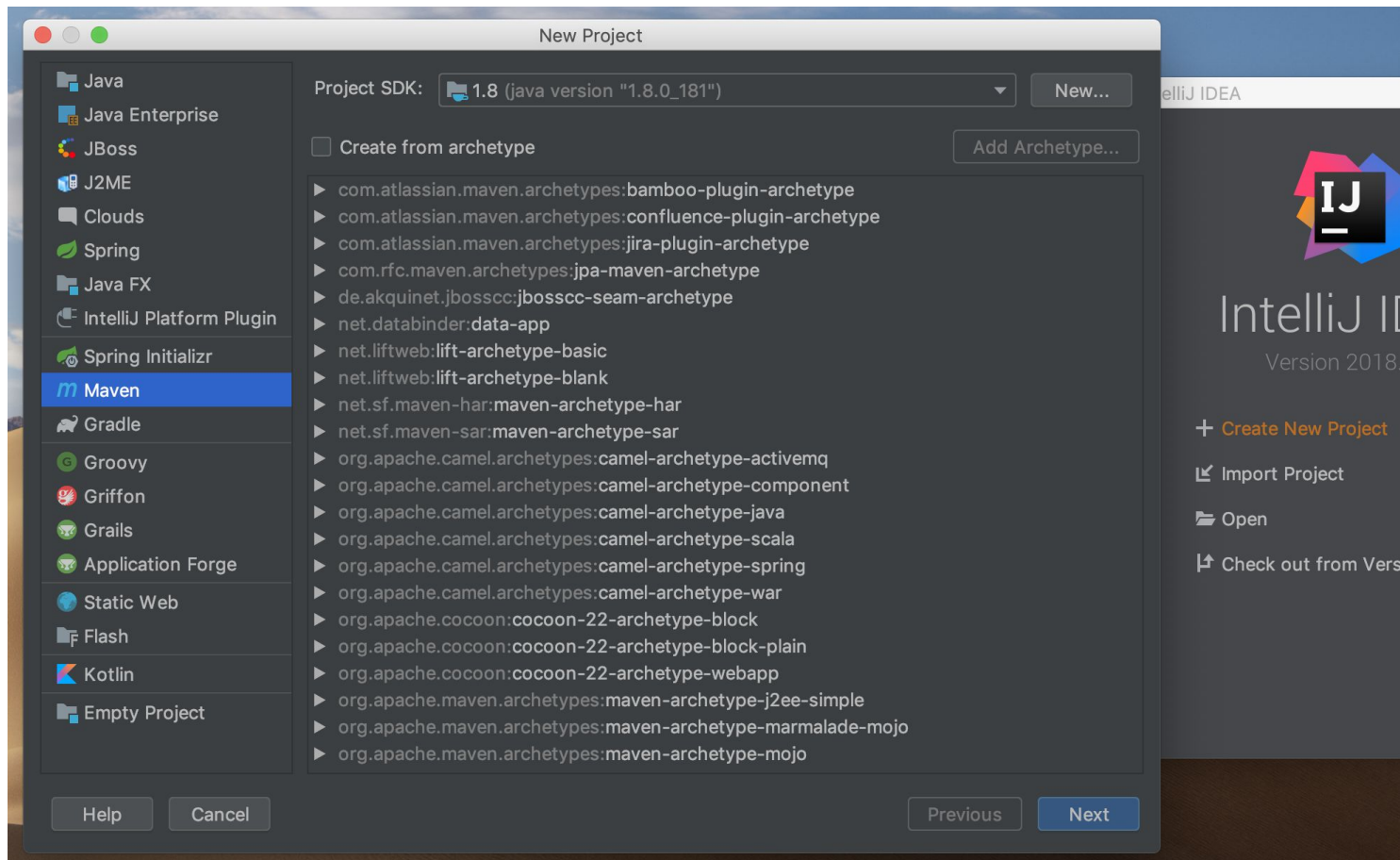
# IntelliJ IDEA

<https://www.jetbrains.com/idea>



# Proyecto Maven

<https://maven.apache.org>



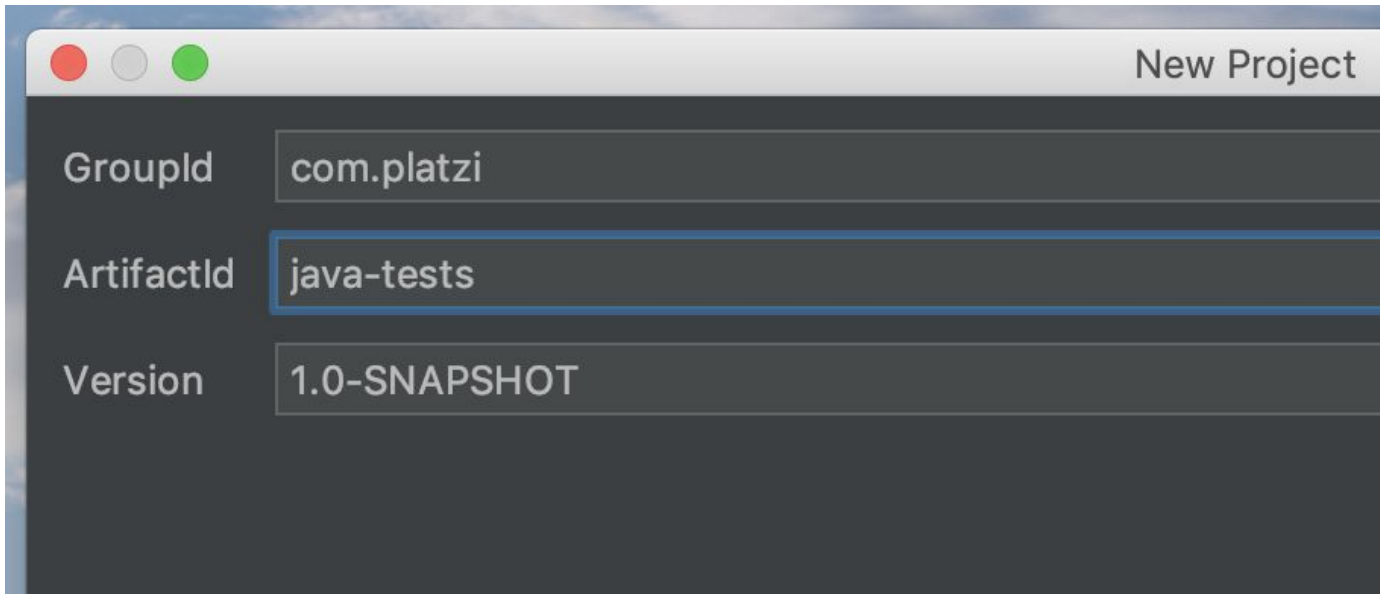
---

# Proyecto Maven

**groupId:** ID de la empresa

**artifactId:** ID del proyecto

**version:** versión



The screenshot shows a 'New Project' dialog box with three input fields. The first field is labeled 'GroupId' and contains the text 'com.platzi'. The second field is labeled 'ArtifactId' and contains the text 'java-tests'. The third field is labeled 'Version' and contains the text '1.0-SNAPSHOT'. The dialog box has a title bar with three colored buttons (red, yellow, green) and the text 'New Project'.

Field	Value
GroupId	com.platzi
ArtifactId	java-tests
Version	1.0-SNAPSHOT

---

# Tests automáticos

Opciones (de peor a mejor):

1. Imprimir el resultado
2. Mostrar OK/Error
3. Sólo mostrar Error
4. Lanzar excepción si hay error
5. Función `assertEquals`

---

# Partes de un test

1. **Given:** preparación del escenario
2. **When:** llamada al método o métodos que queremos probar
3. **Then:** comprobación de que el resultado es el esperado



---

# Librerías: JUnit, Mockito

```
<dependencies>

  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
  </dependency>

  <dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>2.23.4</version>
  </dependency>

</dependencies>
```

---

# Test con JUnit

Comprobamos que el resultado sea el esperado.

```
public class StringUtilRepeatTest {  
  
    @Test  
    public void repeat_text_several_times() {  
        assertThat(  
            repeat(times: 3, text: "ha"),  
            is(value: "hahaha") );  
    }  
  
    // ...  
}
```

---

# Test con Mockito

Simulamos objetos para que retornen el valor que queremos.

```
@Test
public void gambler_wins() {

    Dice dice = mock(Dice.class);
    given(dice.roll()).willReturn(5);

    Gambler g = new Gambler(minNumToWin: 4, dice);

    assertThat(g.play(), is(value: true));
}
```

---

# Análisis y mejoras en los tests

- Uso de @Before
- Nomenclatura de los tests
- Resultados a comprobar

---

# Uso de @Before

- Se ejecuta antes de cada test
- Moveremos a ahí las inicializaciones comunes

```
@Before
public void init() {
    dice = mock(Dice.class);
    gambler = new Gambler(minNumToWin: 4, dice);
}
```

---

# Nomenclatura de los tests

- Clase: *MyClassShould*
- Método: *do\_this\_when\_that()*
- Describir reglas de negocio, no aspectos técnicos ni detalles de implementación

---

# **Tipos de resultados a comprobar**

1. Valor
2. Estado
3. Interacción

---

# Tipos de resultados a comprobar

1. Valor
2. Estado
3. Interacción

```
public class StringUtilRepeatTest {  
  
    @Test  
    public void repeat_text_several_times() {  
        assertThat(  
            repeat( times: 3, text: "ha"),  
            is( value: "hahaha" ) );  
    }  
  
    // ...  
}
```



---

# Tipos de resultados a comprobar

1. Valor
- 2. Estado**
3. Interacción

```
@Test
public void apply_specified_discount() {

    calculator.setDiscount(0.10);
    calculator.addAmounts(20.0, 30.0);

    assertThat(calculator.getTotal(), is(value: 45.0));
}
```

---

# Tipos de resultados a comprobar

1. Valor
2. Estado
3. Interacción

```
@Test
public void use_the_dice() {

    given(dice.roll()).willReturn(5); // given

    gambler.play(); // when

    then(dice).should(times(1)).roll(); // then
}
```

---

# Test Driven Development (TDD)

- Definición, beneficios
- El ciclo red-green-refactor
- Las 3 reglas del TDD

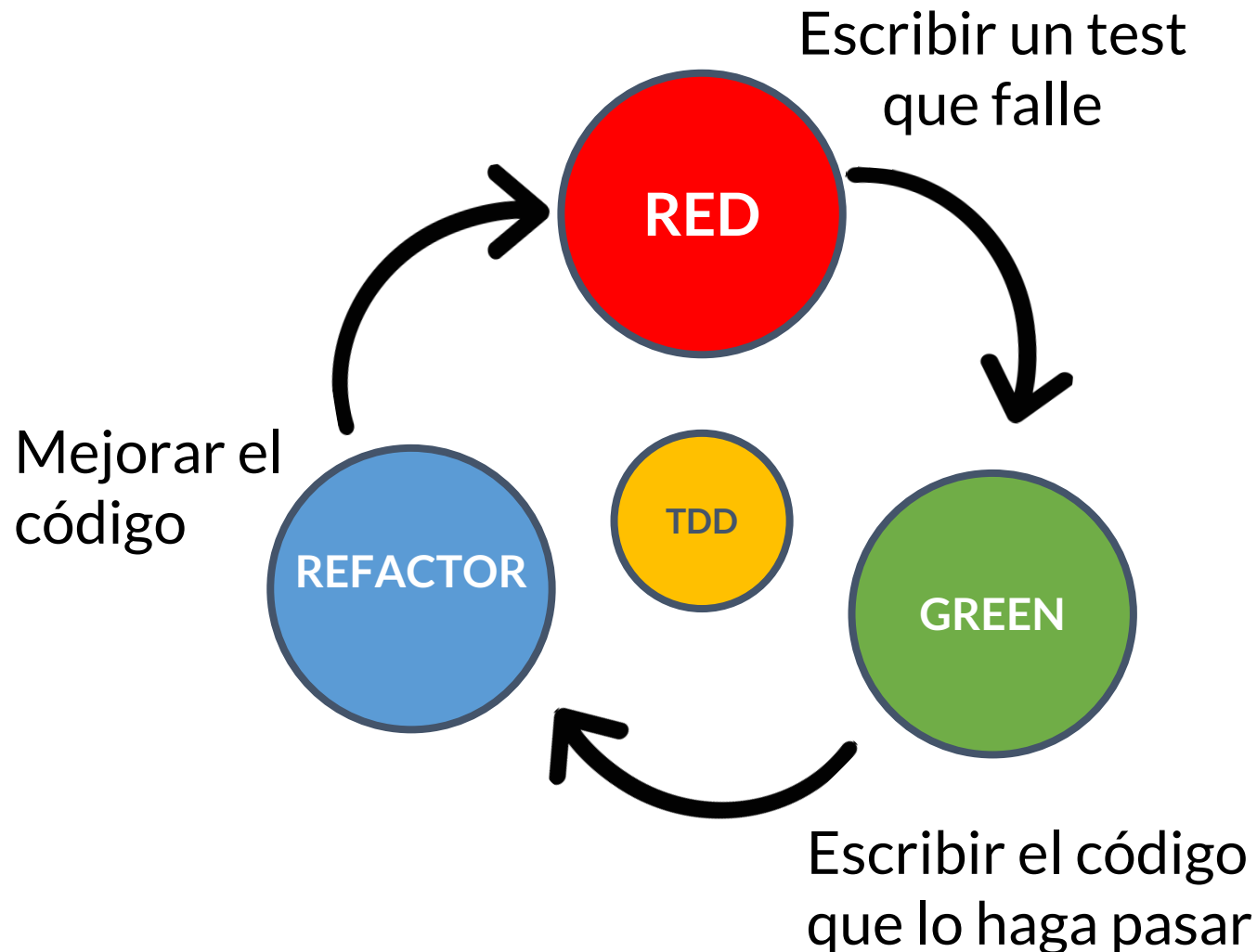
---

# TDD: Definición y Beneficios

- Desarrollo guiado por tests
- Creado por Kent Beck
- Comenzar escribiendo tests nos ayuda a pensar el diseño
- Si escribir el test es difícil, quizás el diseño no es adecuado
- Al terminar ya tendremos los tests, con todos sus beneficios

---

# El ciclo del TDD



---

# Las 3 reglas del TDD

1. Sólo escribirás código de test hasta que falle
2. Sólo escribirás código de producción si es para pasar un test que falla
3. No escribirás más código de producción del necesario para pasar el test

---

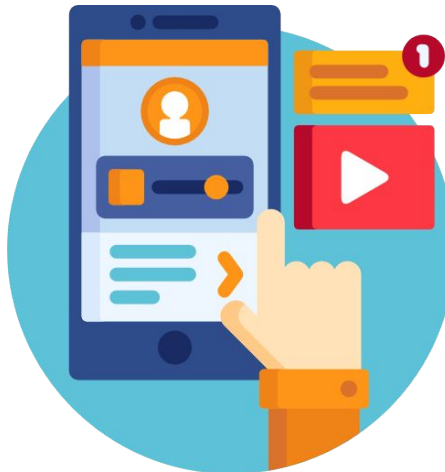
# Las 3 reglas del TDD (mi versión)

1. **Red**: Escribirás el mínimo código de test que falle
2. **Green**: Escribirás el mínimo código de producción que pase el test (reglas originales 2+3)
3. **Refactor**: sólo cuando los tests estén pasando

---

# Tests en una aplicación

- Organización de la aplicación: interfaz, negocio, datos
- Tests de negocio
- Tests de integración (p.ej. BD)

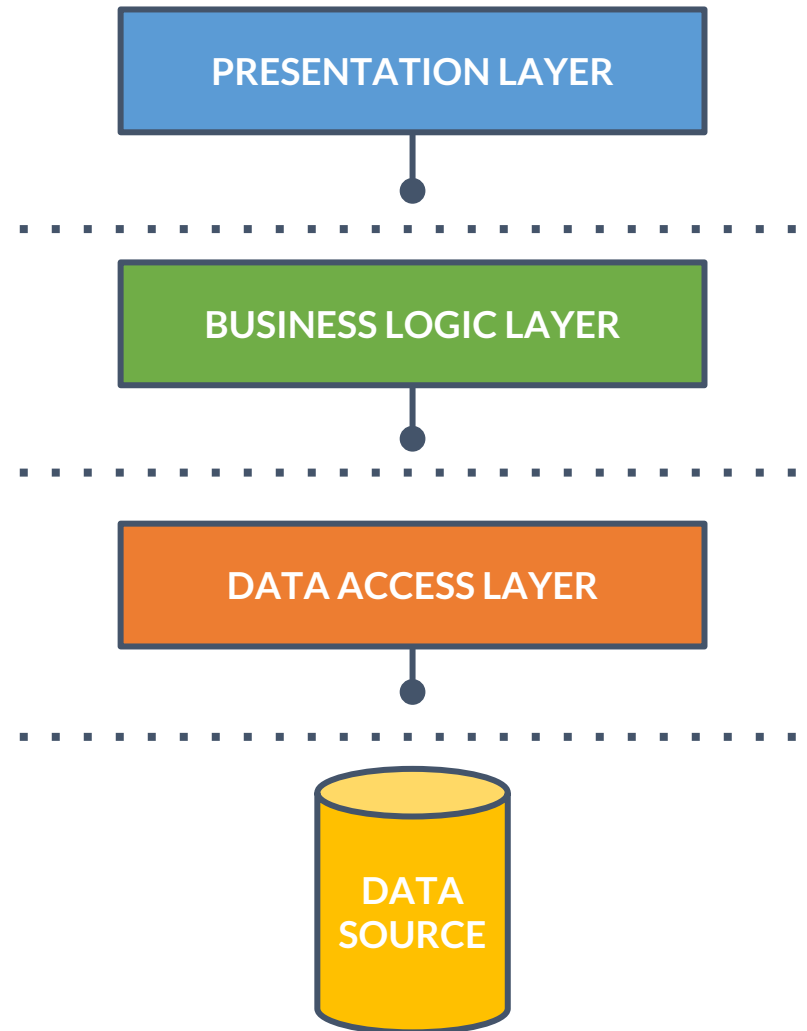




# Organización de la aplicación

Una aplicación generalmente tiene 3 partes/capas:

- Interfaz (parte externa)
- Negocio (lógica de negocio)
- Datos (almacenamiento)



---

# Tests de negocio

- Empezamos por la capa de negocio, que es la más importante
- Probamos que la lógica de negocio es correcta, independientemente de las otras capas
- Podemos hacer mocking de la capa de datos

---

# Tests de integración

- Probamos que las diferentes partes de la aplicación se comunican correctamente
- Por ejemplo, el acceso a BD o el funcionamiento de la API
- Los tests de integración suelen ser más lentos, así que intentaremos tener menos

---

# Requerimientos y tests

- Escribiremos los tests a partir de los requerimientos
- Lo ideal es tener los requerimientos bien especificados
- A veces el programador debe acabar de concretarlos

---

# Tests para diferentes escenarios

- Escenarios típicos (happy path)
- Escenarios extremos
- Escenarios incorrectos
- Escenario no previsto... bug!

---

# Resumen y conclusiones

- Los tests nos aportan muchos beneficios
- Tests a partir de los requerimientos
- Partes: *given, when, then*
- TDD nos puede ayudar en el diseño
- Pensar escenarios (felices, extremos, bugs!)
- Decidir qué tests son necesarios