PRUEBAS UNITARIAS

UNIDAD 4: PRUEBAS DE SOFTWARE

Temario

- Introducción a las pruebas de software
- Verificación & Validación
- TDD
- Pruebas unitarias.
- Pruebas unitarias estáticas y dinámicas.
- JUNIT

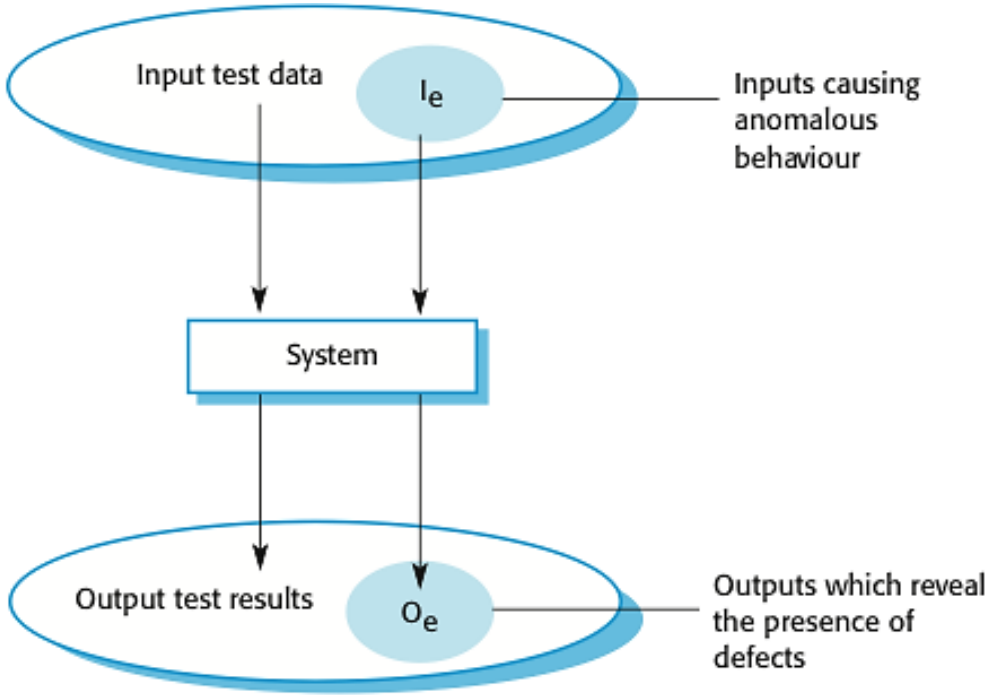
Introducción

Las pruebas tienen como objetivo demostrar que un **programa hace lo que debe hacer** y **descubrir defectos en el programa** antes de ponerlo en uso. Cuando pruebas un software, ejecutas un programa utilizando datos artificiales. Verifica los resultados de la ejecución de prueba en busca de errores, anomalías o información sobre los atributos no funcionales del programa. **Las pruebas pueden revelar la presencia de errores, pero NO su ausencia.** Las pruebas son parte de un proceso de verificación y validación más general, que también incluye técnicas de validación estática.

Objetivos de las pruebas de software:

- Demostrar al desarrollador y al cliente que el **software cumple con sus requisitos** .
  - Conduce a **pruebas de validación** : se espera que el sistema funcione correctamente utilizando un conjunto determinado de casos de prueba que reflejan el uso esperado del sistema.
  - Una prueba exitosa muestra que el sistema funciona según lo previsto.
- Descubrir situaciones en las que el comportamiento del software es **incorrecto , indeseable o no se ajusta a sus especificaciones** .
  - Conduce a **pruebas de defectos** : los casos de prueba están diseñados para exponer defectos; los casos de prueba pueden ser deliberadamente oscuros y no necesitan reflejar cómo se usa normalmente el sistema.
  - Una prueba exitosa es una prueba que hace que el sistema funcione incorrectamente y, por lo tanto, expone un defecto en el sistema.

Objetivos de las pruebas de software:



UNIVERSIDAD DE LIMA

Verificación y validación

- Las pruebas son parte de un proceso más amplio de **verificación y validación** de software (V & V).
  - **Verificación: ¿Estamos construyendo el producto correctamente?**
    El software debe ajustarse a sus especificaciones.
  - **Validación: ¿Estamos construyendo el producto correcto?**
    El software debe hacer lo que el usuario realmente requiere.
- El **objetivo de V&V** es generar confianza en que el sistema es **lo suficientemente bueno para el uso previsto** , lo cual depende de:
  - **Propósito del software** : el nivel de confianza depende de qué tan crítico sea el software para una organización.
  - **Expectativas del usuario** : los usuarios pueden tener bajas expectativas sobre ciertos tipos de software.
  - **Entorno de marketing** : llevar un producto al mercado temprano puede ser más importante que encontrar defectos en el programa.

Inspecciones y pruebas

Las inspecciones de software implican que personas examinen la representación fuente con el objetivo de descubrir anomalías y defectos. Las inspecciones no requieren la ejecución de un sistema por lo que pueden usarse antes de la implementación. Se pueden aplicar a cualquier representación del sistema (requisitos, diseño, datos de configuración, datos de prueba, etc.). Se ha demostrado que son una técnica eficaz para descubrir errores de programas.

Las ventajas de las inspecciones incluyen:

- Durante las pruebas, los errores pueden enmascarar (ocultar) otros errores. Debido a que la inspección es un proceso estático, no es necesario preocuparse por **las interacciones entre errores** .
- **Las versiones incompletas** de un sistema se pueden inspeccionar sin costes adicionales. Si un programa está incompleto, entonces es necesario desarrollar arneses de prueba especializados para probar las piezas disponibles.
- Además de buscar defectos en el programa, una inspección también puede considerar **atributos de calidad más amplios** de un programa, como el cumplimiento de estándares, la portabilidad y la mantenibilidad.

Inspecciones y pruebas

Las inspecciones y pruebas son técnicas de verificación complementarias y no opuestas. Ambos deben usarse durante el proceso V & V. Las inspecciones pueden comprobar la conformidad con una especificación, pero no la conformidad con los requisitos reales del cliente. Las inspecciones no pueden comprobar características no funcionales como rendimiento, usabilidad, etc.

Normalmente, un sistema de software comercial debe pasar por **tres etapas de prueba** :

- **Pruebas de desarrollo** : el sistema se prueba durante el desarrollo para descubrir errores y defectos.
- **Pruebas de lanzamiento** : un equipo de pruebas separado prueba una versión completa del sistema antes de lanzarlo a los usuarios.
- **Pruebas de usuario** : los usuarios o usuarios potenciales de un sistema prueban el sistema en su propio entorno.

Pruebas de desarrollo

Las pruebas de desarrollo incluyen todas las actividades de prueba que lleva a cabo el equipo que desarrolla el sistema:

- **Pruebas unitarias** : se prueban unidades de programas individuales o clases de objetos; debe centrarse en probar la funcionalidad de objetos o métodos.
- **Prueba de componentes** : se integran varias unidades individuales para crear componentes compuestos; debería centrarse en probar las interfaces de los componentes.
- **Prueba del sistema** : algunos o todos los componentes de un sistema se integran y el sistema se prueba en su conjunto; debería centrarse en probar las interacciones de los componentes.

Pruebas Unitarias

Las pruebas unitarias son el proceso de **probar componentes individuales de forma aislada** . Es un proceso de prueba de defectos. Las unidades pueden ser:

- Funciones o métodos **individuales dentro de un objeto;**
- **Clases de objetos** con varios atributos y métodos;
- **Componentes compuestos** con interfaces definidas que se utilizan para acceder a su funcionalidad.

Al **probar clases de objetos** , las pruebas deben diseñarse para brindar cobertura de todas las características del objeto:

- Probar **todas las operaciones** asociadas con el objeto;
- Establecer y verificar el **valor de todos los atributos** asociados con el objeto;
- Poner el objeto en **todos los estados posibles** , es decir, simular todos los eventos que provocan un cambio de estado.

Pruebas de componentes

Los componentes de software suelen ser componentes compuestos que se **componen de varios objetos que interactúan** . Usted accede a la funcionalidad de estos objetos a través de la **interfaz del componente definido** . Por lo tanto, las pruebas de componentes compuestos deberían centrarse en mostrar que la interfaz del componente se comporta de acuerdo con su especificación. Los objetivos son detectar fallos debidos a errores de interfaz o suposiciones no válidas sobre las interfaces. Los tipos de interfaz incluyen:

- **Interfaces de parámetros** : datos pasados de un método o procedimiento a otro.
- **Interfaces de memoria compartida** : el bloque de memoria se comparte entre procedimientos o funciones.
- **Interfaces de procedimiento** : el subsistema encapsula un conjunto de procedimientos que serán llamados por otros subsistemas.
- **Interfaces de paso de mensajes** : los subsistemas solicitan servicios de otros subsistemas.

Pruebas del sistema

Las pruebas del sistema durante el desarrollo implican **la integración de componentes** para crear una versión del sistema y luego **probar el sistema integrado** . El objetivo de las pruebas de sistemas es **probar las interacciones entre los componentes** . Las pruebas del sistema verifican que los componentes sean compatibles, interactúen correctamente y transfieran los datos correctos en el momento adecuado a través de sus interfaces. Las pruebas de sistemas prueban el **comportamiento emergente** de un sistema.

Durante las pruebas del sistema, los componentes reutilizables que se han desarrollado por separado y los sistemas disponibles en el mercado pueden integrarse con componentes recientemente desarrollados. Luego se prueba el sistema completo. En esta etapa se pueden integrar componentes desarrollados por diferentes miembros del equipo o subequipos. La prueba del sistema es un proceso colectivo más que individual.

Pruebas de lanzamiento

La prueba de lanzamiento es el proceso de **probar una versión particular** de un sistema que está **destinado a ser utilizado fuera del equipo de desarrollo** . El objetivo principal del proceso de prueba de lanzamiento es **convencer al cliente del sistema de qué es lo suficientemente bueno para su uso** . Por lo tanto, las pruebas de lanzamiento deben demostrar que el sistema ofrece la funcionalidad, el rendimiento y la confiabilidad especificados y que no falla durante el uso normal. Las pruebas de lanzamiento suelen ser un proceso de prueba de caja negra en el que **las pruebas solo se derivan de la especificación del sistema** .

Pruebas de lanzamiento

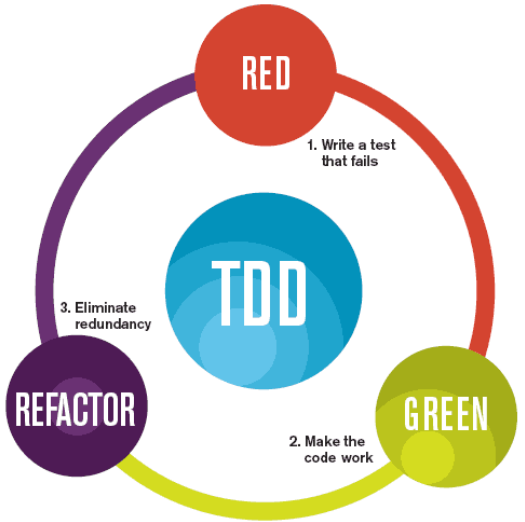
Pruebas de usuario

Las pruebas de usuarios o clientes son una etapa del proceso de pruebas en la que **los usuarios o clientes brindan información y asesoramiento sobre las pruebas del sistema** . Las pruebas de usuario son esenciales, incluso cuando se han realizado pruebas exhaustivas del sistema y de la versión. Los tipos de pruebas de usuario incluyen:

- **Prueba alfa** : los usuarios del software trabajan con el equipo de desarrollo para probar el software en el sitio del desarrollador.
- **Prueba beta** : se pone a disposición de los usuarios una versión del software para permitirles experimentar y plantear los problemas que descubran a los desarrolladores del sistema.
- **Pruebas de aceptación** : los clientes prueban un sistema para decidir si está listo o no para ser aceptado por los desarrolladores del sistema e implementado en el entorno del cliente.

Test-Driven Development

El desarrollo basado en pruebas (TDD) es un enfoque para el desarrollo de programas en el que se **intercalan pruebas y desarrollo de código** .



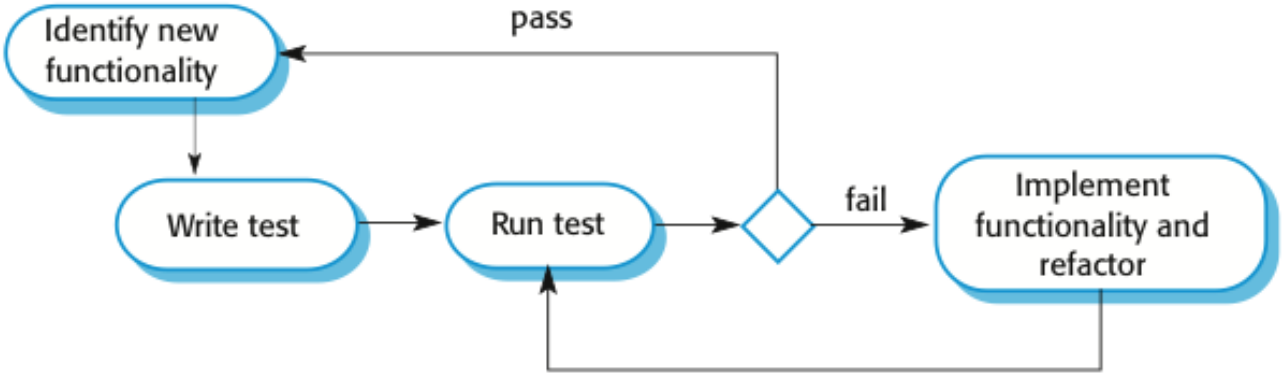
Las pruebas se escriben antes que el código y "aprobarlas" es el motor fundamental del desarrollo. Esta es una característica diferenciadora de TDD frente a la escritura de pruebas unitarias después de escribir el código: hace que el desarrollador se centre en los requisitos antes de escribir el código.

El código se desarrolla de forma incremental, junto con una prueba para ese incremento . No pasa al siguiente incremento hasta que el código que ha desarrollado pase la prueba. TDD se introdujo como parte de métodos ágiles como Extreme Programming. Sin embargo, también se puede utilizar en procesos de desarrollo basados en planes.

UNIVERSIDAD DE LIMA

Test-Driven Development

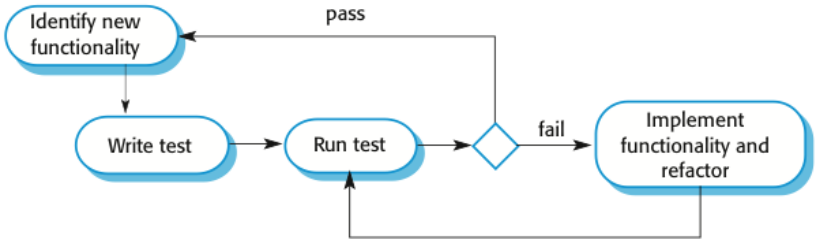
El **objetivo de TDD** no es garantizar que escribimos pruebas escribiéndolas primero, sino producir **software funcional que cumpla con un conjunto específico de requisitos utilizando soluciones simples y fáciles de mantener** . Para lograr este objetivo, TDD proporciona estrategias para **mantener el código funcionando, simple, relevante y libre de duplicaciones** .



Test-Driven Development

El proceso TDD incluye las siguientes actividades:

1. Comience por **identificar el incremento de funcionalidad** que se requiere. Normalmente, esto debería ser pequeño e implementable en unas pocas líneas de código.
2. **Escriba una prueba** para esta funcionalidad e impleméntela como una prueba automatizada.
3. **Ejecute la prueba** , junto con todas las demás pruebas que se hayan implementado. Inicialmente, no implementó la funcionalidad, por lo que la nueva prueba fallará.
4. **Implemente la funcionalidad y vuelva a ejecutar la prueba** .
5. Una vez que todas las pruebas se hayan ejecutado correctamente, pasará a implementar la siguiente parte de la funcionalidad.



UNIVERSIDAD DE LIMA

UNIVERSIDAD DE LIMA | 60 años

Pruebas de Unidad (Unitarias)

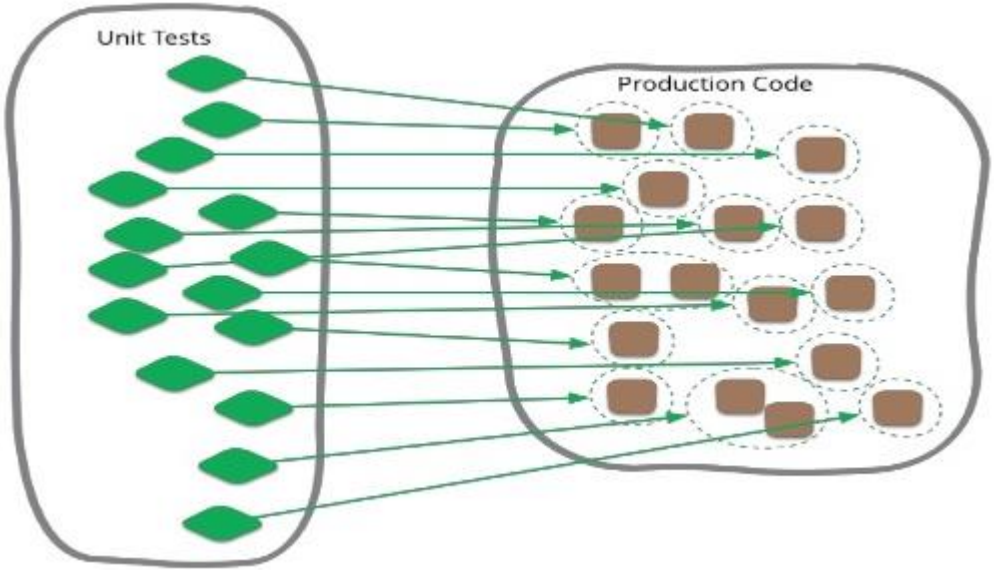
Pruebas Unitarias

Las pruebas unitarias se preocupan por probar programas/clasesindividuales para garantizar que cada programa/clase se ejecute según las especificaciones

- Cada pieza de código funcional (clase o método) será susceptible de ser probada con pruebas unitarias.
- Estas son pruebas automáticas (código) que se encargan de simular la interacción de entidades externas con la pieza de código (desde ahora componente).

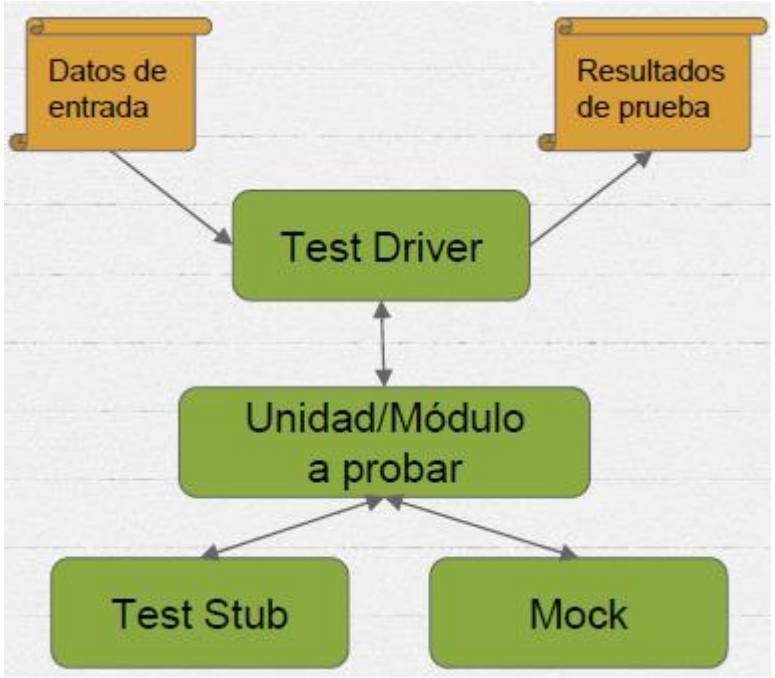
TIPOS DE PRUEBAS

- **Pruebas de Unidad (Unitarias)**
  - Se debe probar el objeto por completo.
    - Todos los métodos
    - Todos las salidas
    - Todas las entradas
    - Todos los estados
    - Todas las herencias
  - Se pueden usar las técnicas de **clases equivalentes**.



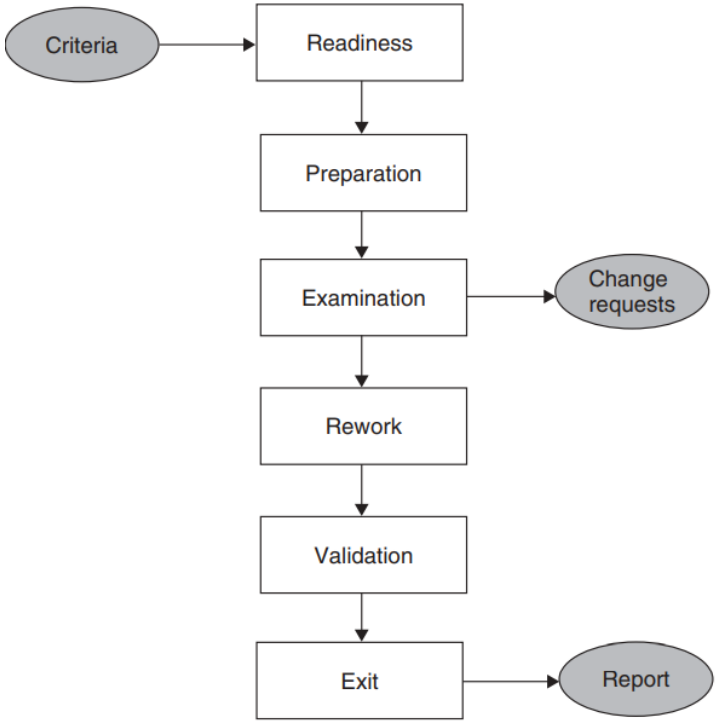
TIPOS DE PRUEBAS

- **Pruebas de Unidad (Unitarias)**
  - **Test Driver (Manejador de prueba)**: Objeto que llama al módulo a probar.
  - **Test Stub**: Objeto (que el módulo a probar llama) que siempre devuelve un resultado estático.
  - **Mock Object:** Objeto (que el módulo a probar llama) que devuelve resultados, sin que esté implementado.



Prueba unitaria estática

- Se llevan a cabo en una **fase de inspección** y corrección donde el producto no necesita estar en su forma final. Por ejemplo, la finalización de la codificación es un hito, pero no necesariamente representa el producto finalizado. La idea detrás de la revisión es encontrar los defectos lo más cerca posible de sus puntos de origen para que esos defectos se eliminen con menos esfuerzo y el producto provisional contenga menos defectos antes de emprender la siguiente tarea.
- Técnicas
  - Inspection
  - Walkthrough



Code review

Prueba unitaria dinámicas

Las pruebas unitarias basadas en la **ejecución** se conocen como pruebas unitarias dinámicas. En esta prueba, una unidad de programa **se ejecuta realmente de forma aislada**.

Se diferencia de la ejecución ordinaria en lo siguiente:

1. Una unidad bajo prueba **se saca de su entorno** de ejecución real.
2. El entorno de ejecución real **se emula** escribiendo más código para que la unidad y el entorno emulado se puedan compilar juntos.
3. La unidad bajo prueba y el código adicional se ejecuta **con entradas seleccionadas previamente**. El resultado de una ejecución de este tipo se recopila de diversas formas y se compara con el resultado esperado. Cualquier diferencia entre el resultado real y el esperado implica una falla y la falla está en el código.

Marco de pruebas unitarias xUnit

Unit es el nombre familiar dado a un grupo de marcos de pruebas unitarias que comparten la misma arquitectura, como JUnit (para Java), NUnit (para .NET), CppUnit (para C++), PHPUnit (para PHP) y muchos otros.

La arquitectura xUnit tiene estos componentes comunes:

- Caso de prueba/conjuntos de pruebas
- Accesorio de prueba
- Corredor de prueba
- Formateador de resultados de prueba
- Afirmaciones:

- **Maven** es una herramienta de software para la gestión y construcción de proyectos **Java**

- Tiene un modelo de configuración de construcción simple, basado en un formato **XML**

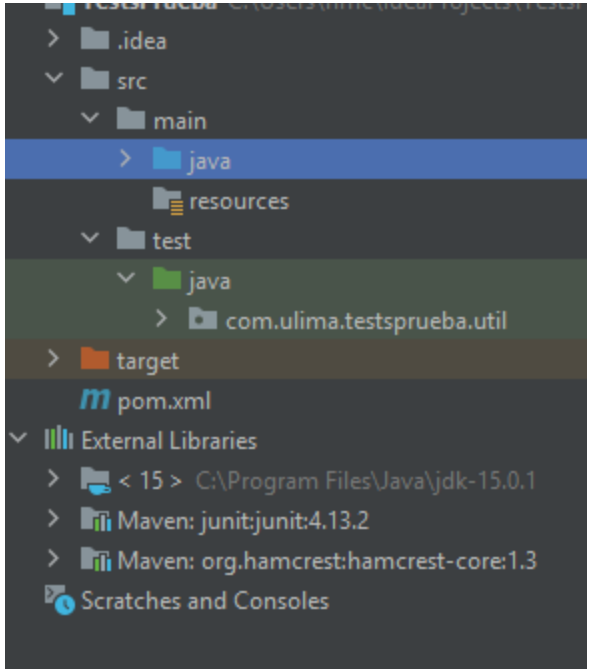
- Maven utiliza un ***Project Object Model*** (**POM**) para describir el proyecto de software a construir, sus dependencias de otros módulos y componentes externos, y el orden de construcción de los elementos

Repositorio oficial de proyectos maven https://mvnrepository.com/

MAVEN

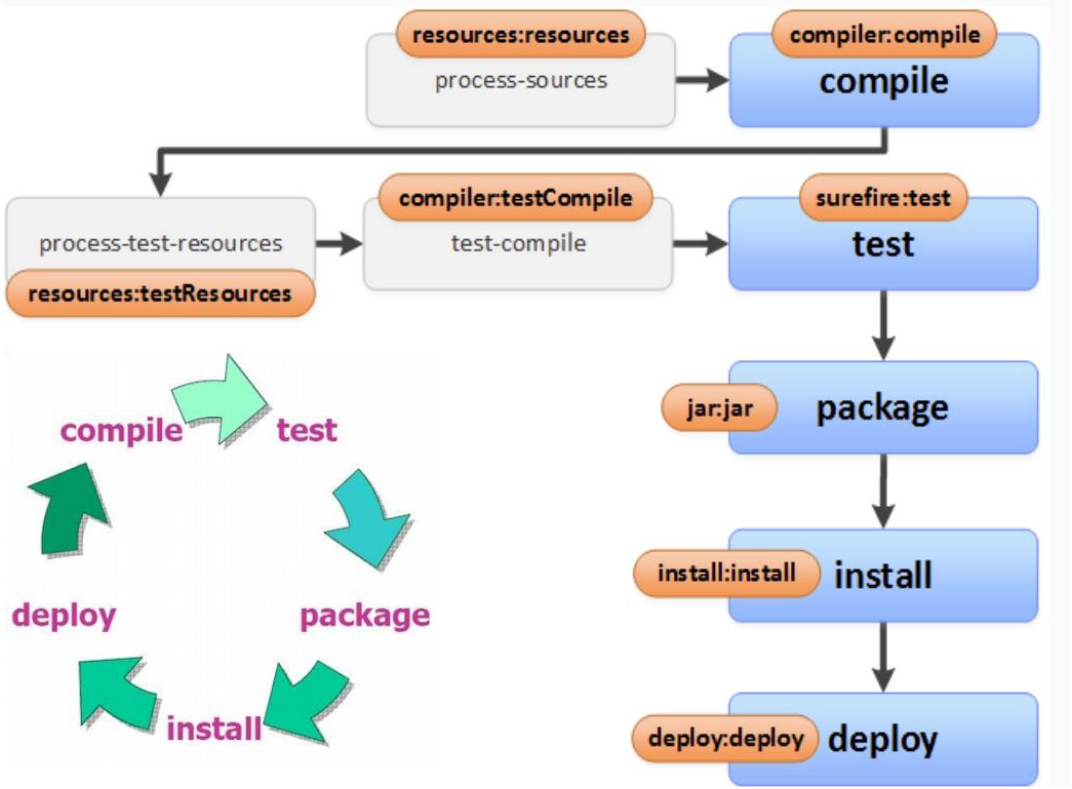
La estructura :

- **src/main/java** : contiene las clases java.
- **src/main/resources** : contiene XMLs, .properties, etc.
- **src/test/java** : contiene las clases java usadas en los tests.
- **src/test/resource**s : contiene XMLs, .properties, etc. usados en los tests.



UNIVERSIDAD DE LIMA

Ciclo de vida del proyecto Maven



UNIVERSIDAD DE LIMA

Estructura de un proyecto- Maven

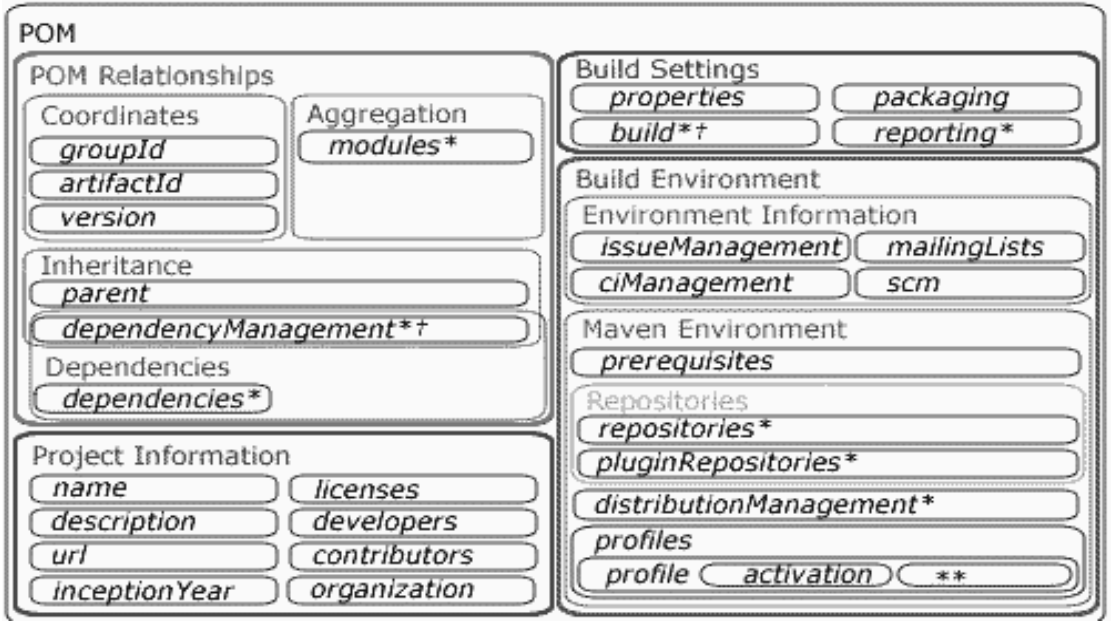
- **groupId:** Aquí se pone el nombre de la empresa u organización, todos los proyectos con ese groupId pertenecen a una sola empresa.
- **artifactId:** Es el nombre del proyecto.
- **version:** Número de versión del proyecto.
- **package:** Paquete base donde irá el código fuente

Opción	Ejemplo
groupId	com.empresa
artifactId	proyecto
version	1.3
package	com.empresa.paquete

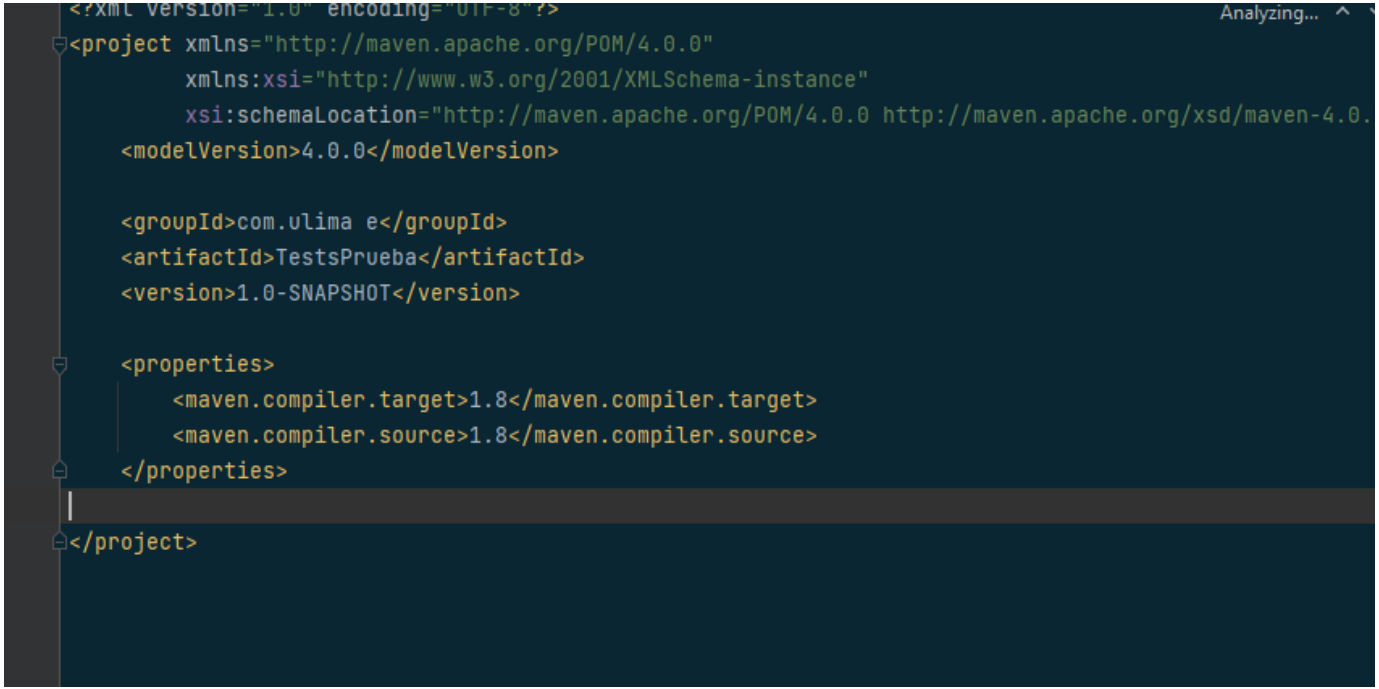
UNIVERSIDAD DE LIMA

Maven

- Esquema de la estructura de un fichero pom.xml



UNIVERSIDAD DE LIMA



UNIVERSIDAD DE LIMA

JUnit

JUnit (@ http://junit.org/) es un *marco de pruebas unitarias* de Java de código abierto diseñado por Kent Beck, Erich Gamma. Es el estándar *de facto* para las pruebas unitarias de Java. JUnit no está incluido en JDK, pero sí en la mayoría de los IDE, como Eclipse y NetBeans.

Las dependencias que necesitamos añadir al pom.xml son:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.8.2</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-test</artifactId>
  <version>3.2.4.RELEASE</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-core</artifactId>
  <version>1.8.5</version>
  <scope>test</scope>
</dependency>
```

JUnit es un paquete Java para automatizar las pruebas de clases Java.
http://junit.org/

UNIVERSIDAD DE LIMA

Junit - @Test

```
import static org.junit.Assert.*;
      import org.junit.Test;

public class Ejemplo {

   @Test
   public void test() {
// prueba
   }
}
```

Los casos de prueba son métodos que

- no devuelven nada: void
- están etiquetados como @Test
- no tienen argumentos ()
- contienen aserciones

UNIVERSIDAD DE LIMA

Junit - Assert

assertEquals (X esperado, Xreal)	compara un resultado esperado con el resultado obtenido, determinando que la prueba pasa si son iguales, y que la prueba falla si son diferentes. Usa el método equals().
assertSame(X esperado, X real)	Ídem; pero usa == para determinar si es el objeto esperado.
assertFalse (boolean resultado)	verifica que el resultado es FALSE
assertTrue (boolean resultado)	verifica que el resultado es TRUE
assertNull (Object resultado)	verifica que el resultado es "null"
assertNotNull (Object resultado)	verifica que el resultado no es "null"
fail	sirve para detectar que estamos en un sitio del programa donde NO deberíamos estar

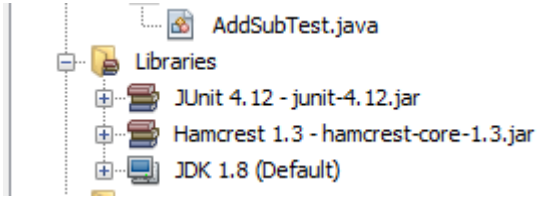
https://junit.org/junit4/javadoc/latest/org/junit/Assert.html

Junit

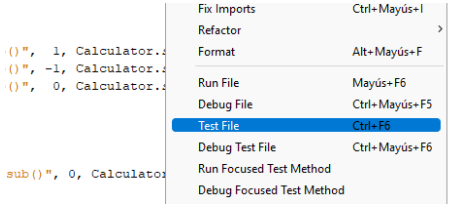
Anotación	*Comportamiento*
`@Before`	El método se ejecutará antes de cada prueba (antes de ejecutar cada uno de los métodos marcados con `@Test`). Será útil para inicializar los datos de entrada y de salida esperada que se vayan a utilizar en las pruebas.
`@After`	Se ejecuta después de cada *test*. Nos servirá para liberar recursos que se hubiesen inicializado en el método marcado con `@Before`.
`@BeforeClass`	Se ejecuta una sola vez antes de ejecutar todos los *tests* de la clase. Se utilizarán para crear estructuras de datos y componentes que vayan a ser necesarios para todas las pruebas. Los métodos marcados con esta anotación deben ser estáticos.
`@AfterClass`	Se ejecuta una única vez después de todos los *tests* de la clase. Nos servirá para liberar los recursos inicializados en el método marcado con `@BeforeClass`, y al igual que este último, sólo se puede aplicar a métodos estáticos.

Ejemplo – Caso de prueba

- Cree un nuevo proyecto Java de llamado " JUnitTest".
- Cree una nueva clase llamada " Calculator"
- Adiciones un JUnit Test y Cree el primer caso de prueba llamado " AddSubTest
- Agregue las librerías



- Ejecute las pruebas



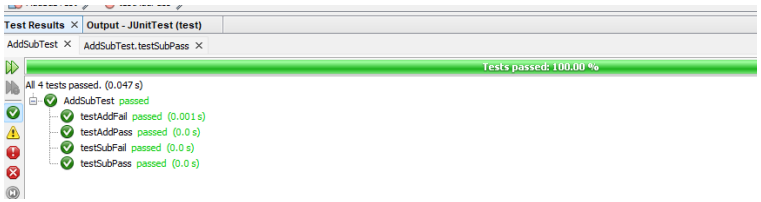
UNIVERSIDAD DE LIMA

Calculator.java

```java
public class Calculator {
    public static int add(int number1, int number2) {
        return number1 + number2;
    }

    public static int sub(int number1, int number2) {
        return number1 - number2;
    }

    public static int mul(int number1, int number2) {
        return number1 * number2;
    }

    public static int divInt(int number1, int number2) {
        if (number2 == 0) {
            throw new IllegalArgumentException("No se puede  dividir por 0!");
        }
        return number1 / number2;
    }

    public static double divReal(int number1, int number2) {
        if (number2 == 0) {
            throw new IllegalArgumentException("No se puede  dividir por 0!");
        }
        return (double) number1 / number2;
    }
}
```

AddSubTest.java

```java
import junittest.Calculator;
import static org.junit.Assert.*;
import org.junit.Test;

public class AddSubTest {
    @Test
    public void testAddPass() {
        assertEquals("error en add()",  3, Calculator.add(1, 2));
        assertEquals("error en add()", -3, Calculator.add(-1, -2));
        assertEquals("error en add()",  9, Calculator.add(9, 0));
    }

    @Test
    public void testAddFail() {
        assertNotEquals("error en add()", 0, Calculator.add(1, 2));
    }

    @Test
    public void testSubPass() {
        assertEquals("error en sub()",  1, Calculator.sub(2, 1));
        assertEquals("error en sub()", -1, Calculator.sub(-2, -1));
        assertEquals("error en sub()",  0, Calculator.sub(2, 2));
    }

    @Test
    public void testSubFail() {
        assertNotEquals("error en sub()", 0, Calculator.sub(2, 1));
    }

}
```



UNIVERSIDAD DE LIMA

Referencias

- Proceso de Construcción de Software 2, Mag. Natalí Flores Lafosse, Maestría en Informática de la PUCP, 2018.
- Sommerville, I. (2005). Ingeniería del software. Pearson educación.
- Bourque, P., & Fairley, R. E. (2014). Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0. IEEE Computer Society Press.