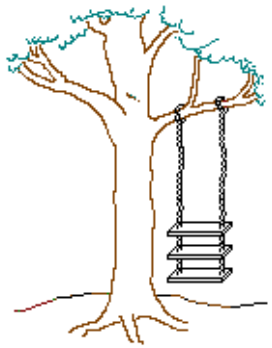
UNIVERSITAT POLITÈCNICA DE VALÈNCIA etsinf

Auditoría, Calidad y Gestión de Sistemas

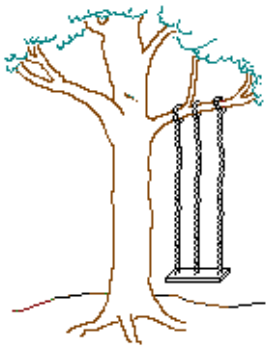
ACG

Tema 2. Estrategia de Testing

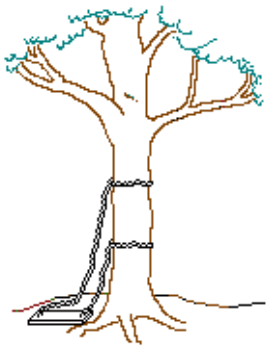
Gestión de la Calidad



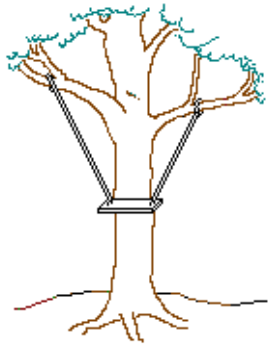
1. Lo que el director desea.



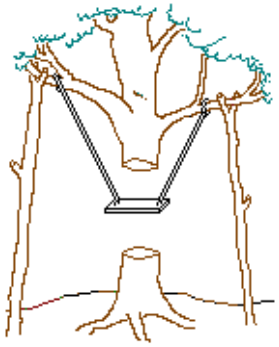
2. Como lo define el director de proyecto.



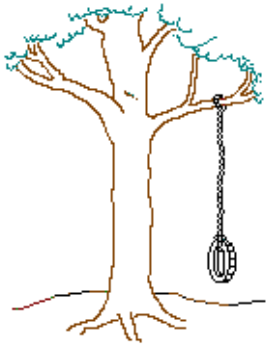
3. Como se diseña el Sistema.



4. Como lo desarrolla el programador.



5. Como se ha realizado la instalación.



6. Lo que el usuario quería.

- El testing es una actividad en la cual un sistema o uno de sus componentes software se ejecuta en **circunstancias previamente especificadas**, los resultados se **observan** y registran y se realiza una **evaluación** de algún aspecto.

- Se dice que una **prueba** ha tenido **éxito** cuando la prueba ha permitido detectar algún error. Por el contrario, la prueba no tiene éxito si no revela ningún error.

Estrategia de testing

Temas en que tenemos que pensar cuando queremos organizar nuestras pruebas de forma estructurada:

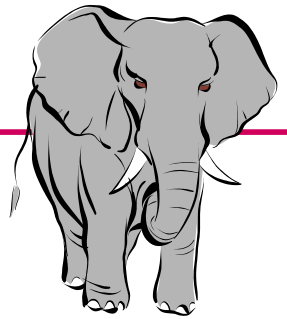
- ¿Niveles?
- ¿Gestión de defectos?
- ¿Qué técnicas?
- ¿Dónde (in-house, outsourcing,..) ?
- ¿Qué herramientas?
- ¿Cuándo empieza el testeo?
- ¿Quién?
- ¿Qué documentación?
- ¿Cobertura?
- ¿Testeo de regresión?
- ¿Métricas?

Cuestiones previas

- ¿Pueden detectar las pruebas la ausencia de errores?
- ¿Se puede realizar una prueba exhaustiva del software (probar todas las posibilidades de su funcionamiento)?

Cuestiones previas

- Limitaciones Teóricas y Prácticas de las Pruebas.
  - Aforismo de Dijkstra:"Probar programas sirve para demostrar la presencia de errores, pero nunca para demostrar su ausencia".
  - En el mundo real no es posible hacer pruebas completas. Se considera que existen infinitos casos de prueba y hay que buscar un equilibrio (recursos y tiempo limitados).



Niveles de testeo

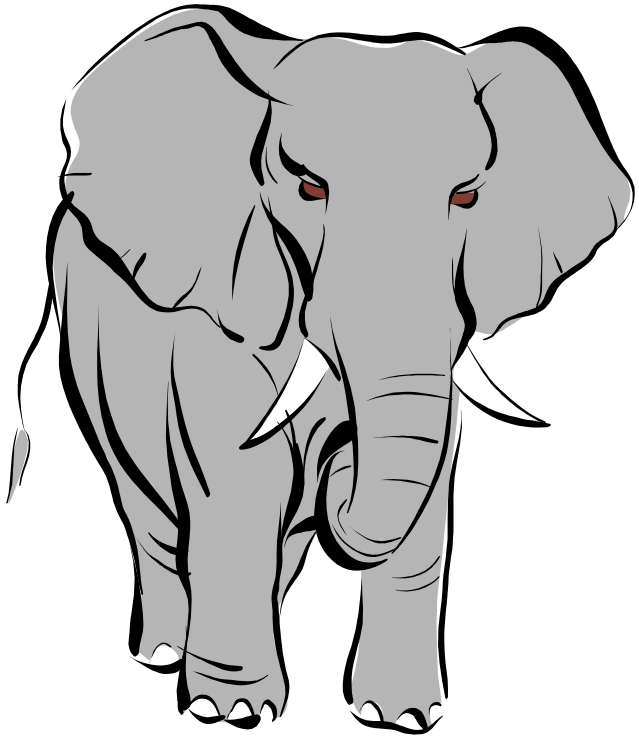
„¿Cómo se come un elefante? … poco a poco“

Proverbio Africano

- Testeo unitario
  - **Testeo de unidades de software individuales**
- Testeo de integración
  - **Testeo de los interfaces de y la interacción entre las unidades previamente testeadas**
- Testeo de sistema
  - **Testeo del sistema entero**
- Testeo de aceptación
  - **Testeo del sistema y los criterios de aceptación previamente establecidos con el cliente**

Parte I. Pruebas Unitarias y de Integración

Testeo Unitario



Testeo Unitario: ¿Qué es?

- **El testing de unidades de software individuales**

- ¿Unidades?
  - Clases/Métodos (en OO)
  - Procedimientos
  - Módulos
  - Componentes
- Como lo defines, depende del:
  - diseño (OO o procedimientos)
  - criticidad del software (más crítico implica unidades más pequeñas)
  - empresa (estrategia)
  - tiempo disponible

Testeo Unitario: ¿Qué es?

- **Prueba Unitaria (Unit Test)**

- Pieza de código escrita por un desarrollador que pone a prueba un pequeño y específico trozo de código.

- Por ejemplo:
  - Tenemos un método que añade un valor a una lista ordenada → construimos un test unitario que compruebe que al ejecutar dicho método el valor añadido aparece en la posición que le corresponde
  - Tenemos un método que borra un patrón de caracteres en un String → construimos un test unitario para comprobar que al llamar a dicho método efectivamente desaparecen los caracteres del patrón de la cadena

Testeo Unitario: ¿Qué es?

- una manera relativamente barata para mejorar la calidad del código producido
- NO es para usuarios, gerentes, jefes de proyectos..
- está hecho por y para programadores
- el programador examina o ejecuta una pequeña parte de su código, para testear:

la funcionalidad que él o ella piensa que tiene que tener

Testeo unitario: Una anécdota

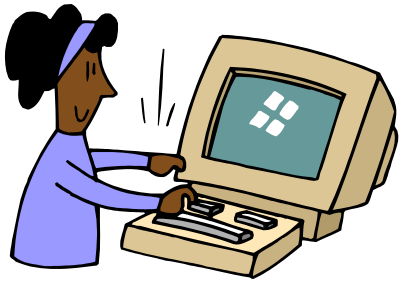
- Juanit@ y Pepit@
  - tienen el mismo plazo de entrega para un paquete
  - la misma edad y experiencia como programador

Testeo unitario: Una anécdota

- Pepit@
  - Produce código rápidamente, la jefa está muy contenta
  - De vez en cuando asegura que su código compila
  - Dos días antes del día de entrega Pepit@ ha terminado y ejecuta todo el código para demostrar que funciona
  - No funciona y Pepit@ utiliza el depurador
  - ¿Cómo puede ser que esta variable sea 0 aquí?
  - Eso es imposible…… ¡J?@$#2/&$!
  - Pepit@ no cumple el plazo de entrega

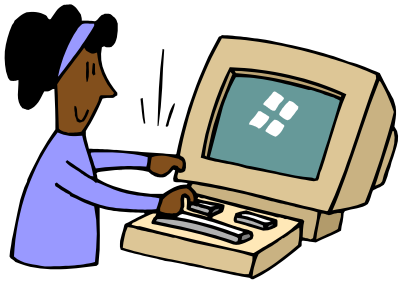
Testeo unitario: Una anécdota

- Juanit@
  - No produce código tan rápido
  - Escribe un test para cada método que produce para ver si el método hace lo que ella quiere que haga
  - Juanit@ termina la noche antes del día de entrega, y ejecuta todo el código para demostrar que funciona
  - Hay una cosita pequeña pero lo puede arreglar fácilmente en 2 minutos
  - Ni siquiera tiene que utilizar el depurador
  - Juanit@ cumple la fecha de entrega



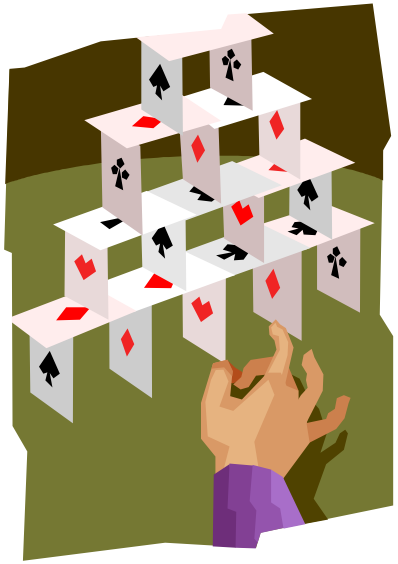
Testeo unitario: Una anécdota

- La diferencia entre Pepit@ y Juanit@:
  - Juanit@ cree en y aplica el testeo unitario



¿Por qué hacer testeo unitario?

- Resulta en mejor código, menos errores
- Reduce el tiempo dedicado a depurar
- Pepit@
  - estaba construyendo un castillo de naipes
  - no podía confiar en su código
  - difícil encontrar los errores
- Juanit@
  - constantemente estaba preguntándose ¿qué es lo que hace mi código? ¿hace lo que espero?
  - podía confiar en su código
  - fácil encontrar errores



Testeo unitario, frutas, verduras y deporte

- Todo el mundo está de acuerdo:

  ¡Tenemos que hacer más testeo!

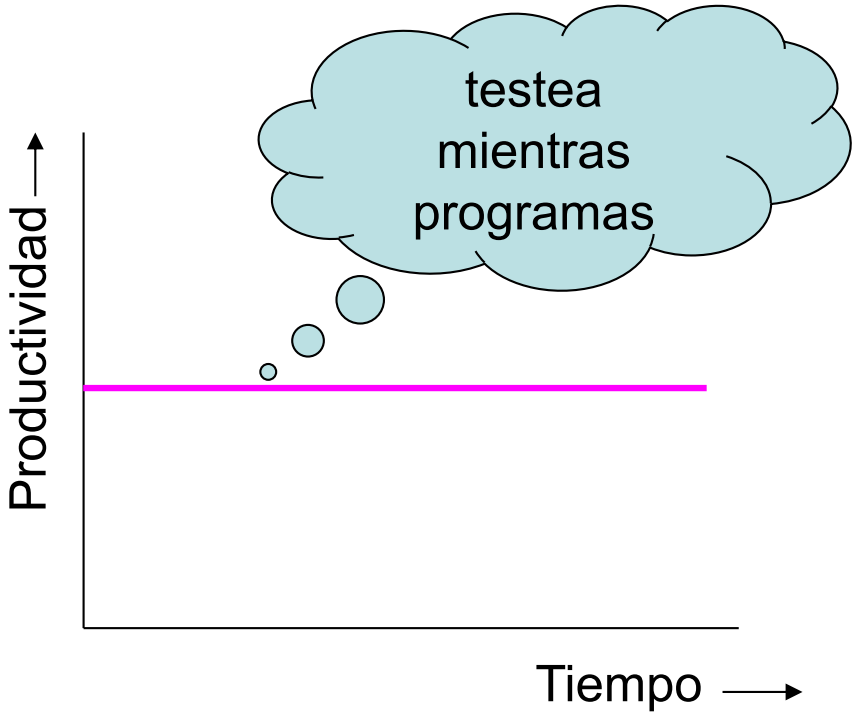
- Pocos realmente lo hacen

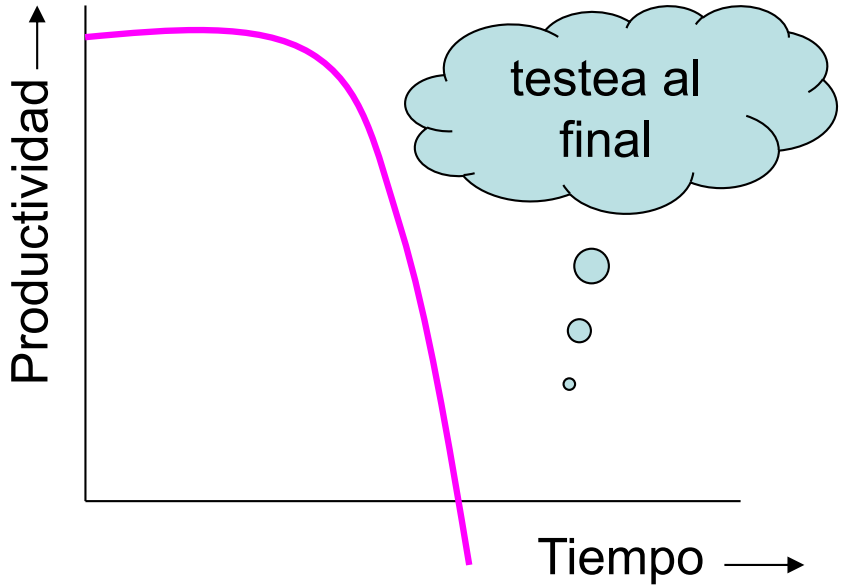
- Muchos utilizan varias excusas porque para ellos es imposible hacer el testeo unitario

Excusas para no hacer testeo unitario

El testeo unitario cuesta mucho tiempo

- Si lo pospones hasta al final cuesta mucho tiempo





Excusas para no hacer testeo unitario

No es mi trabajo hacer testeo, tenemos un departamento de calidad

- Entonces ¿cuál es el trabajo de un programador?
- El trabajo de cada programador es producir código que funciona y que no contenga muchos errores
- Es imposible hacer eso sin testear

Me pagan para escribir código, no tests

- Aplicando este razonamiento tampoco te pagan para estar utilizando el depurador todo el día!

Excusas para no hacer testeo unitario

No sé qué es lo que tiene que hacer el código y entonces no puedo testearlo

- Si no sabes eso no tienes que estar programando tampoco!

¡No hay excusa para no hacer tests unitarios!

- una manera relativamente barata para mejorar la calidad del código producido
- hecho por y para programadores
- el programador examina o ejecuta una pequeña parte de su código, para testear:

la funcionalidad que él o ella piensa que tiene que tener

Testeo Unitario: Herramientas

- Existen diferentes herramientas para agilizar el proceso de pruebas de software unitarias.
  - JUnit para el lenguaje Java
  - NUnit para todos los lenguajes .NET
  - xUnit.NET se lanza posteriormente a NUnit con lecciones aprendidas de Nunit

Testeo Unitario: Herramientas

- ¿Qué formas tenemos de realizar pruebas al software?
  - Con el depurador.
  - Con sentencias que se muestren en la salida estándar.
- ¿Qué problemas presentan estas formas de prueba?
  - Requieren de la participación del *tester* para analizar resultados.
  - No son amigables (sólo puedes ejecutar una expresión del depurador a la vez, multitud de sentencias mostrando expresiones de prueba).
- Beneficios de las herramientas de prueba:
  - permite escribir código más rápidamente e incrementa su calidad
  - es simple
  - comprueba sus propios resultados y proporcionan feedback inmediato

- **JUnit**

  - JUnit es un framework para el desarrollo y ejecución de unidades de prueba utilizando el lenguaje Java
  - Es *Open Source*
  - JUnit permite escribir pruebas repetibles
  - Es una instancia de la arquitectura xUnit para frameworks de pruebas unitarias
  - Plugins para multiples entornos de desarrollo

- **JUnit**
  - Métodos `@Test`
    - Son los métodos de prueba
  - Métodos `@BeforeEach, @BeforeAll, @AfterEach, @AfterAll`
    - `@BeforeEach` se invoca antes de la ejecución de **cada** test
    - `@AfterEach` se invoca después de la ejecución de **cada** test
    - `@BeforeAll` se invoca antes de la ejecución de **todos** los tests
    - `@AfterAll` se invoca después de la ejecución de **todos** los tests
  - Sentencias `assert`
    - `assertEquals`
    - `assertNotNull`
    - `assertTrue`
    - `assertSame`
    - `...`

Simple JUnit Example

```
public class Lista {
    public Lista(String[] elementos){…}
    public Lista ordenar(){…}
}
```


Simple JUnit Example

```
public class Lista {
    public Lista(String[] elementos){…}
    public Lista ordenar(){…}
}
```


Simple JUnit Example

- Posible caso de prueba:

```
String[] cadena = {“e”, “d”, “c”, “b”, “a”};
Lista reves = new Lista(cadena);
Lista ordenada = reves.ordenar();
```

Y el resultado esperado:

“a”,”b”,”c”,”d”,”e”

Simple JUnit Example

```
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;
```

- Con JUnit 5.0:

Anotación @Test que indica que es un método de prueba

```
@Test
public void testOrdenarReves(){
    String[] cadenaOrdenada = {"a","b","c","d","e"};
    String[] cadenaDesordenada = {"b","e","a","d","c"};
    Lista expected = new Lista(cadenaOrdenada);
    Lista listaAlReves = new Lista(cadenaDesordenada);
    assertEquals(expected, listaAlReves.ordenar());
}
```

Método para evaluar resultados

Simple JUnit Example

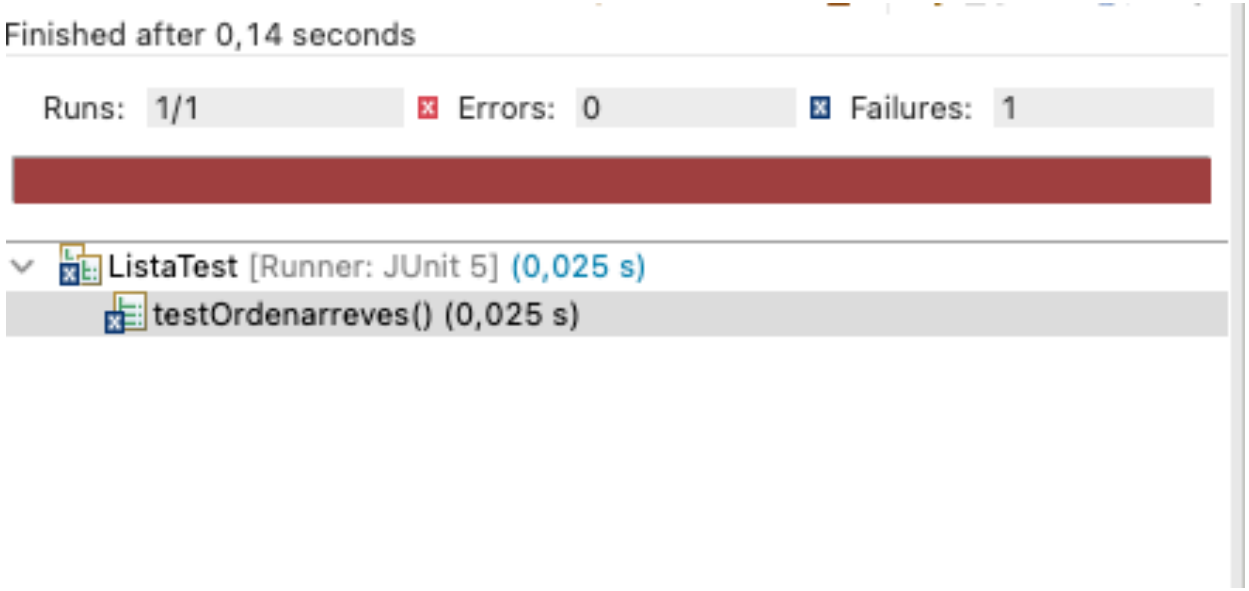
- ¿Dónde ponemos este código?:

Una clase de prueba por cada clase a probar

```
public class ListaTest {

  @Test
  public void testOrdenarReves(){
      String[] cadenaOrdenada = {"a","b","c","d","e"};
            String[] cadenaDesordenada = {"b","e","a","d","c"};
      Lista expected = new Lista(cadenaOrdenada);
      Lista listaAlReves = new Lista(cadenaDesordenada);
      assertEquals(expected, listaAlReves.ordenar());
     }
}
```


Simple JUnit Example



- **JUnit**
  - Métodos `@Test`
    - Son los métodos de prueba
  - Métodos `@BeforeEach, @BeforeAll, @AfterEach, @AfterAll`
    - `@BeforeEach` se invoca antes de la ejecución de **cada** test
    - `@AfterEach` se invoca después de la ejecución de **cada** test
    - `@BeforeAll` se invoca antes de la ejecución de **todos** los tests
    - `@AfterAll` se invoca después de la ejecución de **todos** los tests
  - Sentencias `assert`
    - `assertEquals`
    - `assertNotNull`
    - `assertTrue`
    - `assertSame`
    - `...`

Simple JUnit Example

```
public class TestDB {
        static private Connection dbConn;
        static private Account acc;
        @BeforeAll
        public static void setUpBeforeAll(){
                dbConn = new Connection("Oracle",15,"fred", "f");
                dbConn.connect();
        }
        @AfterAll
        public static void tearDownAfterAll(){
                dbConn.disconnect();
                dbConn = null;
        }
        @BeforeEach
        protected void setUp(){
                acc = new Account();
        }
        @AfterEach
        protected void tearDown(){
                acc = null;
        }
```

```
public void testAccountAccess(){
        …/Uses dbConn and acc
}

public void testEmployeeAccess(){
        …/Uses dbConn and acc
}
```


Simple JUnit Example

- Al ejecutar las pruebas el orden es el siguiente

```
setUpBeforeAll
   setUp
       testAccountAccess
   tearDown
   setUp
        testEmployeeAccess
   tearDown
 tearDownAfterAll
```


Test Suits JUnit

- JUnit ofrece la posibilidad de definir un **Test suite**.
- Un test suite es una combinación de clases de prueba que permite ejecutar todas las pruebas contenidas en dichas clases a la vez en un orden establecido.
- Para crear una Test suit se debe:
  - Crear una clase Java
  - Anotar la clase con la etiqueta **@RunWith(JUnitPlatform.class)**
  - Indicar las clases que forman parte de la suite con la etiqueta **@SelectClasses**, o **@SelectPackages**

Test Suits JUnit

A continuación, se muestra un ejemplo de código de la **test suite** `AllTests` que agrupa las pruebas de las clases `LargestTest` y `MiClaseDeTest`.

```
import org.junit.platform.runner.JUnitPlatform;
import org.junit.platform.suite.api.SelectClasses;
import org.junit.runner.RunWith;

@RunWith(JUnitPlatform.class)
@SelectClasses({ LargestTest.class, MiClaseDeTest.class })
public class AllTests {

}
```

Cuando lancemos a ejecución el **test suite** `AllTests` se ejecutarán primero las pruebas de la clase `LargestTest`, y a continuación las pruebas de la clase `MiClaseDeTest`.

Tests parametrizados

- Los **test parametrizados** posibilitan ejecutar un test con diferentes argumentos.
- Estos tests son declarados como los tests básicos (@Test methods) pero usan la etiqueta **@ParameterizedTest**.
- Es necesario declarar al menos una fuente que proporcionará los argumentos para cada invocación del método.
  - Esto se puede hacer con la anotación @ValueSource y un array de valores literales.
  - Existen también otras formas de hacerlo, por ejemplo, con valores de un fichero **csv** utilizando la anotación **@CsvFileSource**.

Tests parametrizados

El siguiente ejemplo muestra un test parametrizado que utiliza la anotación `@ValueSource` para especificar una cadena de strings con los argumentos para el método de prueba.

```
@ParameterizedTest
@ValueSource(strings = { "racecar", "radar", "able was I
ere I saw elba" })
void palindromesTest(String candidate) {
    assertTrue(StringUtils.isPalindrome(candidate));
}
```


Tests parametrizados

El siguiente ejemplo muestra un test parametrizado que utiliza la anotación `@ValueSource` para especificar una cadena de strings con los argumentos para el método de prueba.

```
@ParameterizedTest
@ValueSource(strings = { "racecar", "radar", "able was I
ere I saw elba" })
void palindromesTest(String candidate) {
    assertTrue(StringUtils.isPalindrome(candidate));
}
```

El test tiene un parámetro (candidate que es un String)

Tests parametrizados

El siguiente ejemplo muestra un test parametrizado que utiliza la anotación `@ValueSource` para especificar una cadena de strings con los argumentos para el método de prueba.

```
@ParameterizedTest
@ValueSource(strings = { "racecar", "radar", "able was I
ere I saw elba" })
void palindromesTest(String candidate) {
    assertTrue(StringUtils.isPalindrome(candidate));
}
```

Que se utiliza como argumento al llamar al método bajo prueba

Tests parametrizados

El siguiente ejemplo muestra un test parametrizado que utiliza la anotación `@ValueSource` para especificar una cadena de strings con los argumentos para el método de prueba.

```
@ParameterizedTest
@ValueSource(strings = { "racecar", "radar", "able was I
ere I saw elba" })
void palindromesTest(String candidate) {
    assertTrue(StringUtils.isPalindrome(candidate));
}
```

Cuando se ejecuta el método de test parametrizado, cada invocación se hace por separado. La consola de la ejecución mostrará lo siguiente:

```
palindromesTest(String) ✓
├─ [1] candidate=racecar ✓
├─ [2] candidate=radar ✓
└─ [3] candidate=able was I ere I saw elba ✓
```


Tests parametrizados

```
@ParameterizedTest
@CsvSource({"2,4", "6,8", "10,12"})
void plusNumberTest(int input, int expected) {
        int value= plusNumber(input);
        assertEquals(expected, value);
}

@ParameterizedTest
@CsvSource(value = {"2:4", "6:8", "10:12"}, delimiter = ':')
void plusNumberTest(int input, int expected) {
    int value= plusNumber(input);
    assertEquals(expected, value);
}
```


Tests parametrizados

```
@ParameterizedTest
@CsvSource({"2,4", "6,8", "10,12"})
void plusNumberTest(int input, int expected) {
        int value= plusNumber(input);
        assertEquals(expected, value);
}

@ParameterizedTest
@CsvSource(value = {"2:4", "6:8", "10:12"}, delimiter = ':')
void plusNumberTest(int input, int expected) {
    int value= plusNumber(input);
    assertEquals(expected, value);
}
```


Tests parametrizados

Valores de prueba: input y expected

```
@ParameterizedTest
@CsvSource({"2,4", "6,8", "10,12"})
void plusNumberTest(int input, int expected) {
        int value= plusNumber(input);
        assertEquals(expected, value);
}

@ParameterizedTest
@CsvSource(value = {"2:4", "6:8", "10:12"}, delimiter = ':')
void plusNumberTest(int input, int expected) {
    int value= plusNumber(input);
    assertEquals(expected, value);
}
```


Tests parametrizados

```
@ParameterizedTest
@CsvSource({"2,4", "6,8", "10,12"})
void plusNumberTest(int input, int expected) {
        int value= plusNumber(input);
        assertEquals(expected, value);
}

@ParameterizedTest
@CsvSource(value = {"2:4", "6:8", "10:12"}, delimiter = ':')
void plusNumberTest(int input, int expected) {
    int value= plusNumber(input);
    assertEquals(expected, value);
}
```

Valores de prueba: input y expected

Tests parametrizados

El test tiene dos parámetros: input y expected

```
@ParameterizedTest
@CsvSource({"2,4", "6,8", "10,12"})
void plusNumberTest(int input, int expected) {
        int value= plusNumber(input);
        assertEquals(expected, value);
}

@ParameterizedTest
@CsvSource(value = {"2:4", "6:8", "10:12"}, delimiter = ':')
void plusNumberTest(int input, int expected) {
    int value= plusNumber(input);
    assertEquals(expected, value);
}
```


Tests parametrizados

```
@ParameterizedTest
@CsvSource({"2,4", "6,8", "10,12"})
void plusNumberTest(int input, int expected) {
        int value= plusNumber(input);
        assertEquals(expected, value);
}

@ParameterizedTest
@CsvSource(value = {"2:4", "6:8", "10:12"}, delimiter = ':')
void plusNumberTest(int input, int expected) {
    int value= plusNumber(input);
    assertEquals(expected, value);
}
```


Tests parametrizados

```
@ParameterizedTest
@CsvSource({"2,4", "6,8", "10,12"})
void plusNumberTest(int input, int expected) {
        int value= plusNumber(input);
        assertEquals(expected, value);
}

@ParameterizedTest
@CsvSource(value = {"2:4", "6:8", "10:12"}, delimiter = ':')
void plusNumberTest(int input, int expected) {
    int value= plusNumber(input);
    assertEquals(expected, value);
}
```


Ejercicio factorial

Vamos a probar!!

```
public class Factorial {

  public static int factorial(int n) {
        if (n < 0) {
            throw new IllegalArgumentException("n must be non-
            negative");
        }
        int result = 1;
        for (int i = 1; i <= n; i++) {
            result *= i;
        }
        return result;
  }

}
```

Plantead test unitarios para este código

Vamos a probar!!

```
public class Factorial {

  public static int factorial(int n) {
        if (n < 0) {
            throw new IllegalArgumentException("n must be non-
            negative");
        }
        int result = 1;
        for (int i = 1; i <= n; i++) {
            result *= i;
        }
        return result;
  }

}
```

Ahora plantead un test parametrizado

Ejercicio ecuación de primer grado

Vamos a probar!!

- Partís de las clases:
  - EcuacionPrimerGrado
  - Parseador
- Plantead las pruebas unitarias
  - Pruebas para la clase Parseador
  - Pruebas para la clase EcuacionPrimerGrado

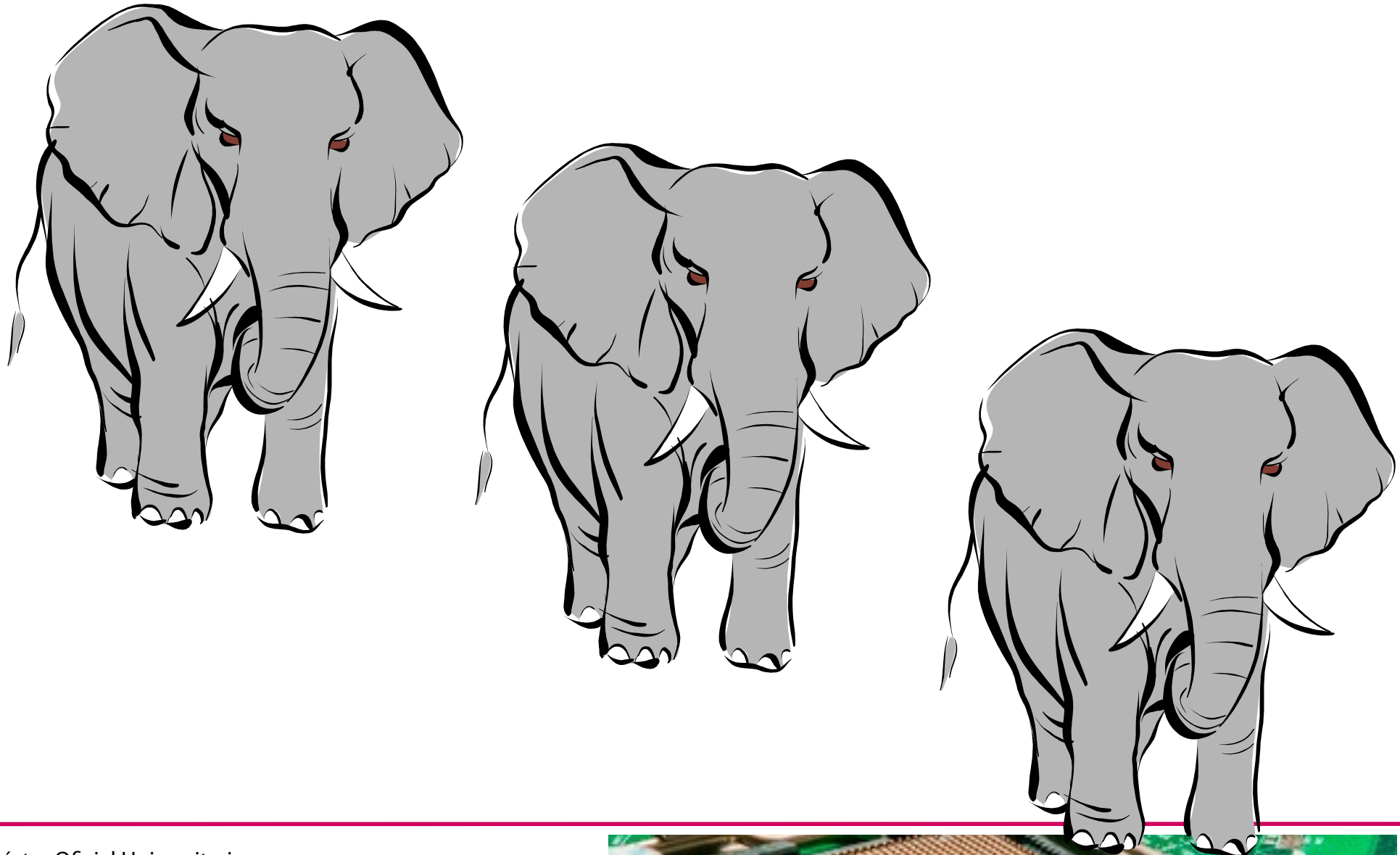
Vamos a probar!!

- Partís de las clases:
  - EcuacionPrimerGrado
  - Parseador
- Plantead las pruebas unitarias
  - **Pruebas para la clase Parseador**
  - Pruebas para la clase EcuacionPrimerGrado

Vamos a probar!!

- Partís de las clases:
  - EcuacionPrimerGrado
  - Parseador
- Plantead las pruebas unitarias
  - Pruebas para la clase Parseador
  - **Pruebas para la clase EcuacionPrimerGrado**

Testeo de integración



Testing de integración

- **Testeo de integración = El testing de las interfaces y la interacción entre las unidades previamente testeadas mientras que se ensambla el sistema entero**

- Una estrategia de testeo de integración tiene que definir:

  1. ¿Qué componentes son el foco del testeo de integración?
  2. ¿En qué orden vamos a testear las interfaces?
  3. ¿Qué técnicas utilizamos para testear las interfaces?

- Depende de la arquitectura del sistema y el enfoque del desarrollo (OO o estructurado).

El foco y el alcance depende de las unidades

Orientada a Objetos

Foco de la integración (Componente)	Alcance de la integración (Sistema)
Método	Clase
Clase	Cluster
Cluster	Subsistema
Subsistema	Sistema

Estructurada

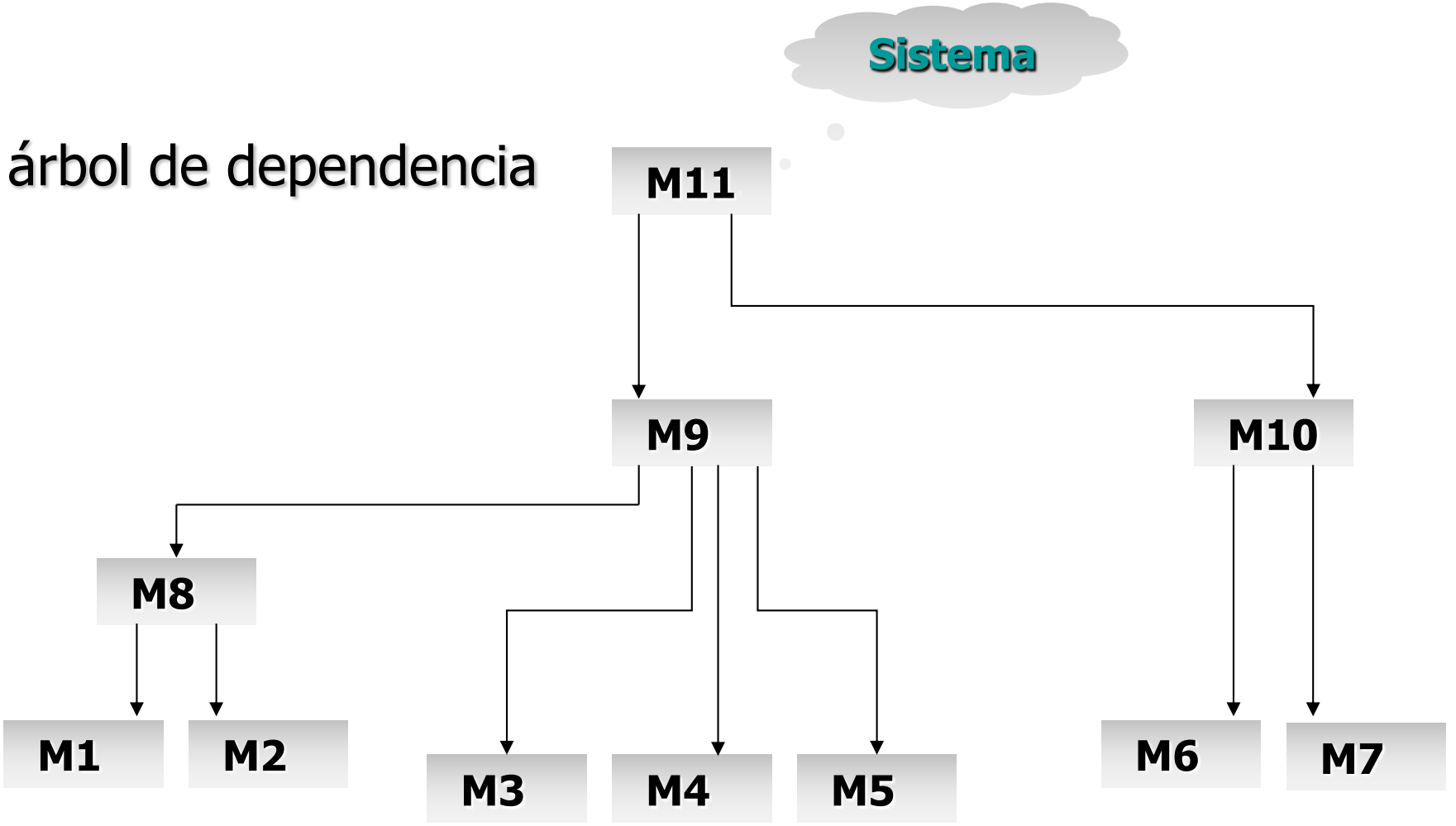
Foco de la integración (Componente)	Alcance de la integración (Sistema)
Función/procedimiento	Módulo
Módulo	Subsistema
Subsistema	Sistema

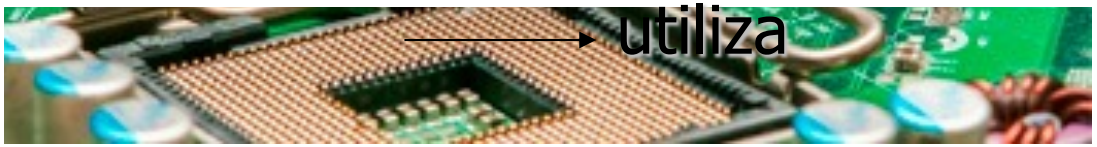
Orden de la integración

- Tres estrategias más descritas:
  - Bottom-up (de abajo arriba)
  - Top-down (de arriba abajo)
  - Big bang

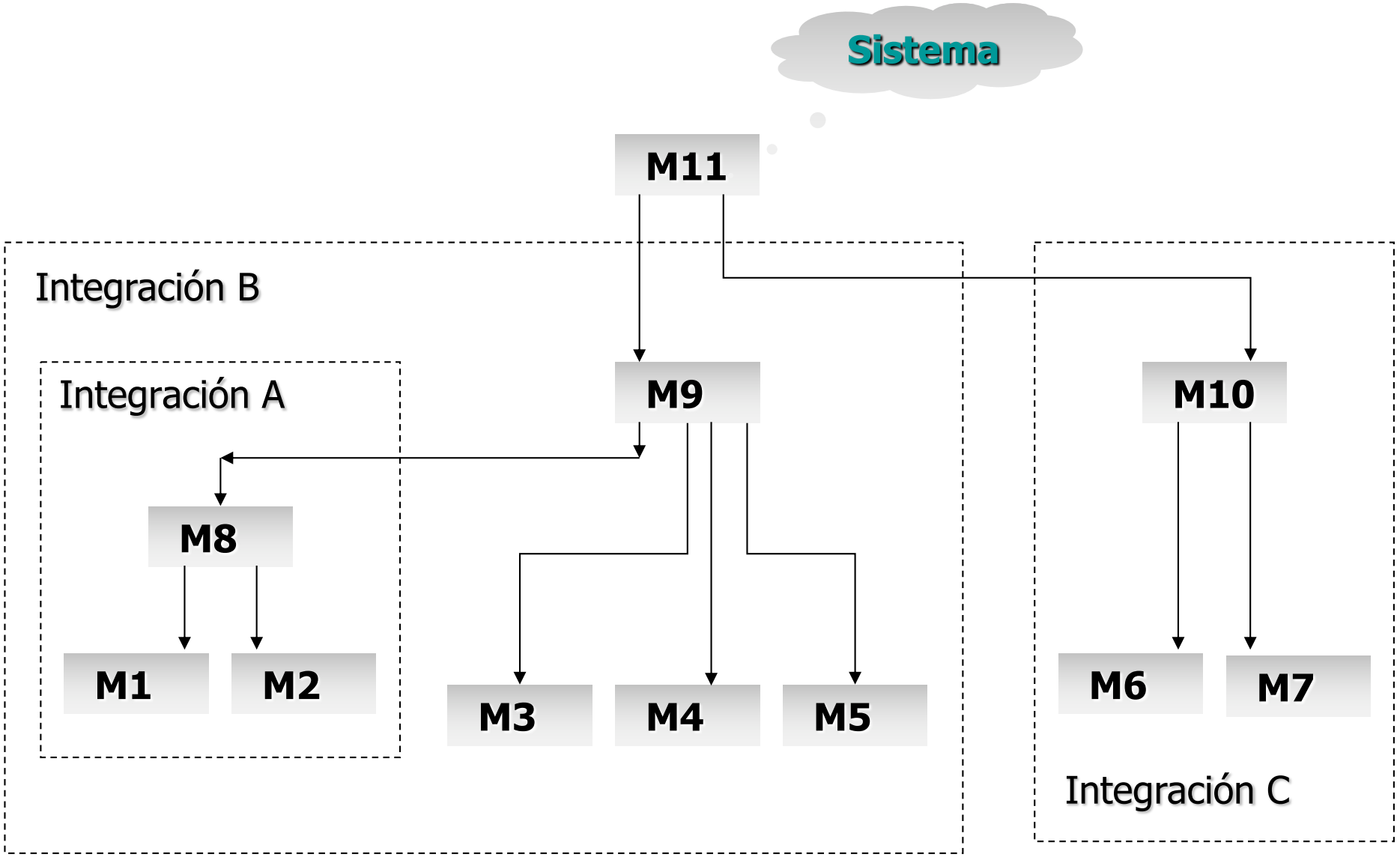
Ejemplo: componentes de un sistema

Un árbol de dependencia



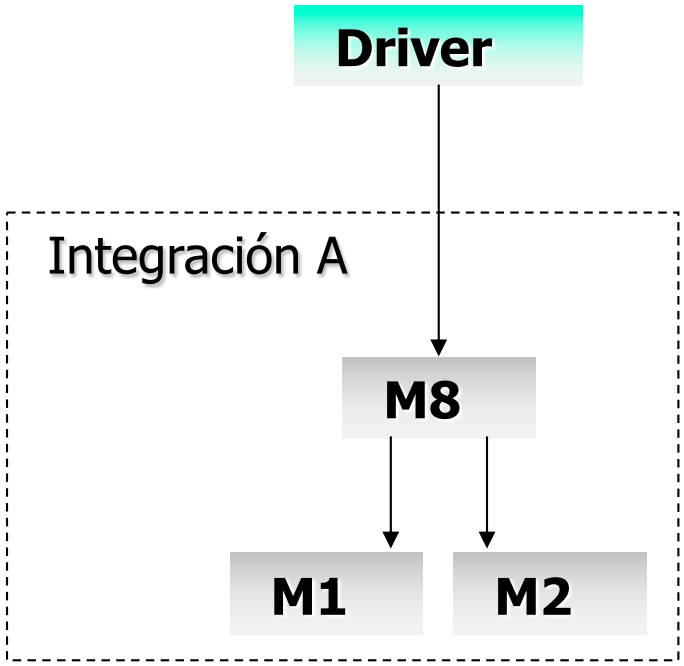


Testing de integración (bottom-up)



Bottom-up: drivers

- Para hacer el testeo de integración de manera bottom-up, se necesita *drivers*
- Un *driver* es un programa que invoca un componente bajo testeo, por ejemplo, para simular un componente de un nivel superior cuyo código todavía no está disponible (está todavía en desarrollo)



Testing de integración (bottom-up)

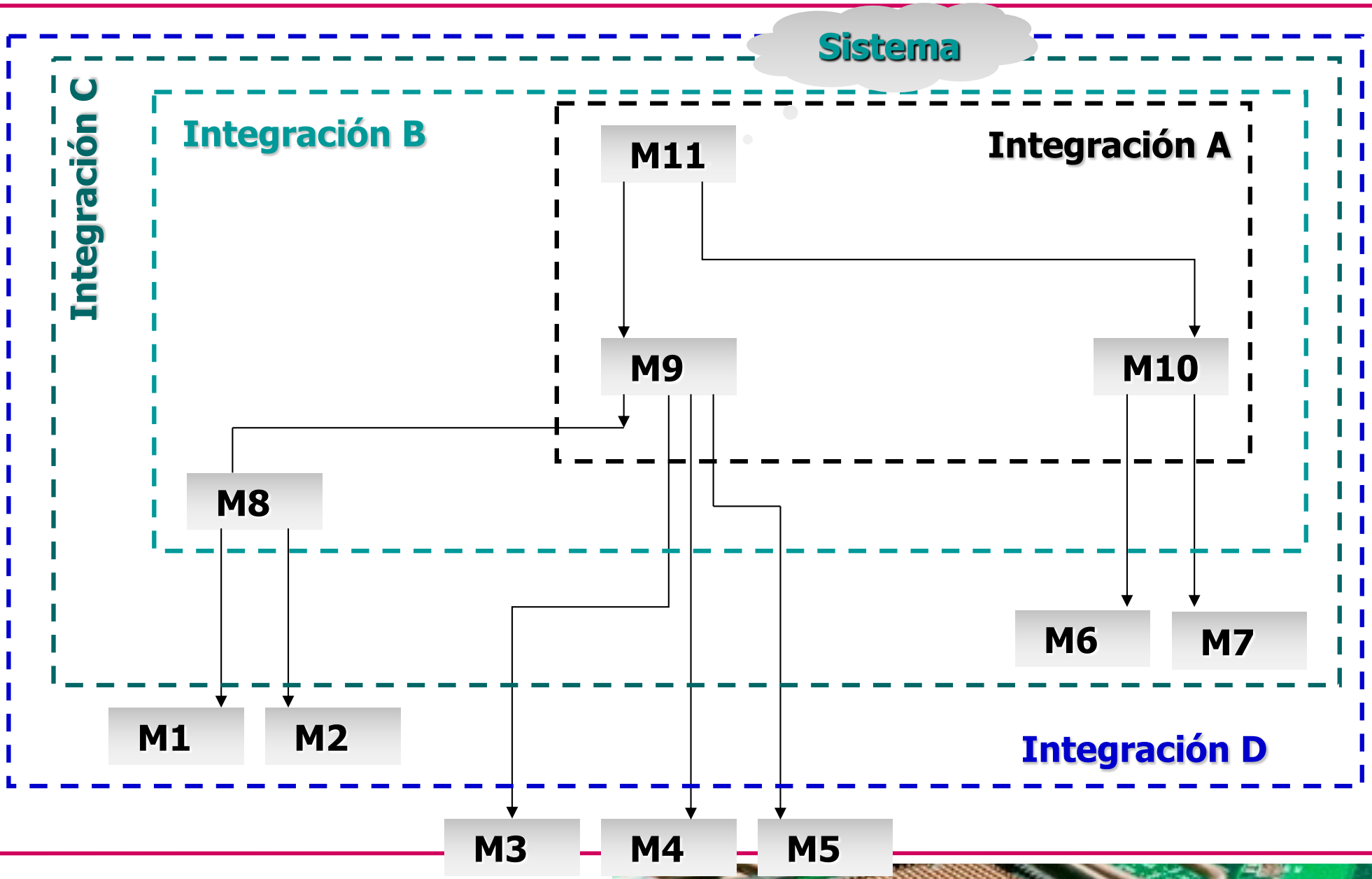
Ventajas

- Relativamente fácil de hacer
- El testeo puede empezar temprano
- Testeo se puede hacer en paralelo

Desventajas

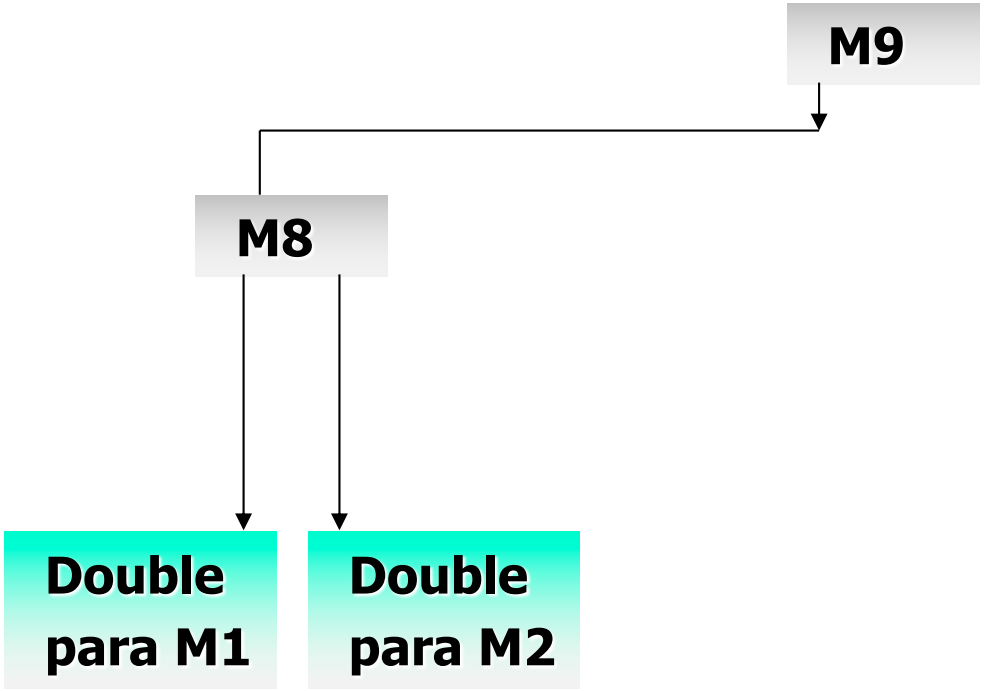
- Visibilidad del sistema tardía
- Desarrollo de *drivers* puede ser muy costoso

Testing de integración (top-down)



Top-down: stubs

- Para hacer el testeo de integración de manera top-down, se necesita *dobles*
- Un *doble* **simula un componente de un nivel inferior**
- Stub, dummy, fake, mock



Top-down: stubs

- ¿Cómo creamos estos dobles?

Top-down: stubs

```
public float calculaSalarioNeto(float salarioBruto) throws BRException {
    float retencion = 0.0f;
    if(salarioBruto < 0) {
        throw new BRException("El salario bruto debe ser positivo"); }
    ProxyAeat proxy = getProxyAeat();
    List<TramoRetencion> tramos;
    try { tramos = proxy.getTramosRetencion();
    } catch (IOException e) {
        throw new BRException("Error al conectar al servidor de la AEAT", e); }
    for(TramoRetencion tr: tramos) {
        if(salarioBruto < tr.getLimiteSalario()) {
            retencion = tr.getRetencion();
            break;
        }
    }
    return salarioBruto * (1 - retencion);
}
```

```
ProxyAeat getProxyAeat() {
    ProxyAeat proxy = new ProxyAeat();
    return proxy;
}
```


Top-down: stubs

```
public class MockProxyAeat extends ProxyAeat {
    boolean lanzarExcepcion;
    public MockProxyAeat(boolean lanzarExcepcion) {
        this.lanzarExcepcion = lanzarExcepcion;
    }
    @Override
    public List<TramoRetencion> getTramosRetencion()
        throws IOException {
            if(lanzarExcepcion) {
            throw new IOException("Error al conectar al servidor");
        }
        List<TramoRetencion> tramos = new ArrayList<TramoRetencion>();
        tramos.add(new TramoRetencion(1000.0f, 0.0f));
        tramos.add(new TramoRetencion(1500.0f, 0.16f));
        tramos.add(new TramoRetencion(Float.POSITIVE_INFINITY, 0.18f));
        return tramos;
    }
}
```


Top-down: stubs

```
ProxyAeat getProxyAeat() {
    ProxyAeat proxy = new ProxyAeat();
    return proxy;
}
```

```
class TestableEmpleado extends Empleado {
    ProxyAeat proxy;

    public void setProxyAeat(ProxyAeat proxy) {
        this.proxy = proxy;
    }

    @Override
    ProxyAeat getProxyAeat() {
        return proxy;
    }
}
```


Top-down: stubs

```
TestableEmpleado empleadoTest;
@BeforeEach
public void setUpClass() {
    empleadoTest = new TestableEmpleado ();
    empleadoTest.setProxyAeat(new MockProxyAeat(false));
    empleadoTestFail = new TestableEmpleado();
    empleadoTestFail.setProxyAeat(new MockProxyAeat(true));
}

@Test
public void testCalculaSalarioNeto1() throws BRException {
    float resultadoReal = empleadoTest .calculaSalarioNeto(2000.0f);
    float resultadoEsperado = 1640.0f;
    assertEquals(resultadoEsperado, resultadoReal, 0.01);
}
```


Testing de integración (top-down)

Ventajas

- Visibilidad del sistema muy temprana

Desventajas

- Difícil de hacer
- Desarrollo de *stubs* puede ser muy costoso

Testing de integración (Big Bang)

- Simplemente ponemos todos los componentes juntos y testeamos el conjunto
- Ventajas:
  - Rápido
  - Simple
- Desventaja
  - Difícil localizar y trazar errores en interfaces
  - No sistemática
- Sólo se puede utilizar cuando
  - El SBT es pequeño
  - El SBT es estable y sólo se añade algunos componentes
  - El SBT es monolítico, hay tantas dependencias entre los componentes que no es factible testearlos por separado

Frameworks para creación de mocks

```
public class CalculatorServiceImpl implements CalculatorService
{
    private DataService dataService;

    public double calculateAverage() {
        int[] numbers = dataService.getListOfNumbers();
        double avg = 0;
        for (int i : numbers) {
            avg += i;
        }
        return (numbers.length > 0) ? avg / numbers.length : 0;
    }
    public void setDataService(DataService dataService) {
        this.dataService = dataService;
    }
}
```

```
public interface DataService {
    int[] getListOfNumbers(); }
public interface CalculatorService {
    double calculateAverage(); }
```


```
@RunWith(MockitoJUnitRunner.class)
public class CalculatorServiceTest {
    @InjectMocks
    private CalculatorServiceImpl calculatorService;
    @Mock
    private DataService dataService;
    @Test
    public void testCalculateAvg_simpleInput() {
        when(dataService.getListOfNumbers()).thenReturn(new int[]{1,2,3,4,5});
        assertEquals(3.0, calculatorService.calculateAverage(), .01);
    }
    @Test
    public void testCalculateAvg_emptyInput() {
        when(dataService.getListOfNumbers()).thenReturn(new int[] {});
        assertEquals(0.0, calculatorService.calculateAverage(), .01);
    }
    @Test
    public void testCalculateAvg_singleInput() {
        when(dataService.getListOfNumbers()).thenReturn(new int[] { 1 });
        assertEquals(1.0, calculatorService.calculateAverage(), .01);
    }
}
```


```
@RunWith(MockitoJUnitRunner.class)
public class CalculatorServiceTest {
    @InjectMocks
    private CalculatorServiceImpl calculatorService;
    @Mock
    private DataService dataService;
    @Test
    public void testCalculateAvg_simpleInput() {
        when(dataService.getListOfNumbers()).thenReturn(new int[]{1,2,3,4,5});
        assertEquals(3.0, calculatorService.calculateAverage(), .01);
    }
    @Test
    public void testCalculateAvg_emptyInput() {
        when(dataService.getListOfNumbers()).thenReturn(new int[] {});
        assertEquals(0.0, calculatorService.calculateAverage(), .01);
    }
    @Test
    public void testCalculateAvg_singleInput() {
        when(dataService.getListOfNumbers()).thenReturn(new int[] { 1 });
        assertEquals(1.0, calculatorService.calculateAverage(), .01);
    }
}
```

Se utiliza para definir que se utilizará el Runner de Mockito para ejecutar nuestras pruebas.

Mockito y JUnit

```
@RunWith(MockitoJUnitRunner.class)
public class CalculatorServiceTest {
    @InjectMocks
    private CalculatorServiceImpl calculatorService;
    @Mock
    private DataService dataService;
    @Test
    public void testCalculateAvg_simpleInput() {
        when(dataService.getListOfNumbers()).thenReturn(new int[]{1,2,3,4,5});
        assertEquals(3.0, calculatorService.calculateAverage(), .01);
    }
    @Test
    public void testCalculateAvg_emptyInput() {
        when(dataService.getListOfNumbers()).thenReturn(new int[] {});
        assertEquals(0.0, calculatorService.calculateAverage(), .01);
    }
    @Test
    public void testCalculateAvg_singleInput() {
        when(dataService.getListOfNumbers()).thenReturn(new int[] { 1 });
        assertEquals(1.0, calculatorService.calculateAverage(), .01);
    }
}
```

Se utiliza para informar a Mockito que los mocks serán inyectados en el servicio definido.

Mockito y JUnit

```
@RunWith(MockitoJUnitRunner.class)
public class CalculatorServiceTest {
    @InjectMocks
    private CalculatorServiceImpl calculatorService;
    @Mock
    private DataService dataService;
    @Test
    public void testCalculateAvg_simpleInput() {
        when(dataService.getListOfNumbers()).thenRetur
        assertEquals(3.0, calculatorService.calculateA
    }
    @Test
    public void testCalculateAvg_emptyInput() {
        when(dataService.getListOfNumbers()).thenReturn(new int[] {});
        assertEquals(0.0, calculatorService.calculateAverage(), .01);
    }
    @Test
    public void testCalculateAvg_singleInput() {
        when(dataService.getListOfNumbers()).thenReturn(new int[] { 1 });
        assertEquals(1.0, calculatorService.calculateAverage(), .01);
    }
}
```

Se utiliza para informar a Mockito que un objeto mock será inyectado en la referencia dataService, como se puede ver no es necesario escribir la clase implementación, Mockito lo hace por nosotros

Mockito y JUnit

```
@RunWith(MockitoJUnitRunner.class)
public class CalculatorServiceTest {
    @InjectMocks
    private CalculatorServiceImpl calculatorService;
    @Mock
    private DataService dataService;
    @Test
    public void testCalculateAvg_simpleInput() {
        when(dataService.getListOfNumbers()).thenReturn(new int[]{1,2,3,4,5});
        assertEquals(3.0, calculatorService.calculateAverage(), .01);
    }
    @Test
    public void testCalculateAvg_emptyInput() {
        when(dataService.getListOfNumbers()).thenReturn(new int[] {});
        assertEquals(0.0, calculatorService.calculateAverage(), .01);
    }
    @Test
    public void testCalculateAvg_singleInput() {
        when(dataService.getListOfNumbers()).thenReturn(new int[] { 1 });
        assertEquals(1.0, calculatorService.calculateAverage(), .01);
    }
}
```

Define que cuando se ejecute el método getListOfNumbers se devolverá el resultado indicado (new int[], { …})

Vamos a probar!!

Vamos a probar!!