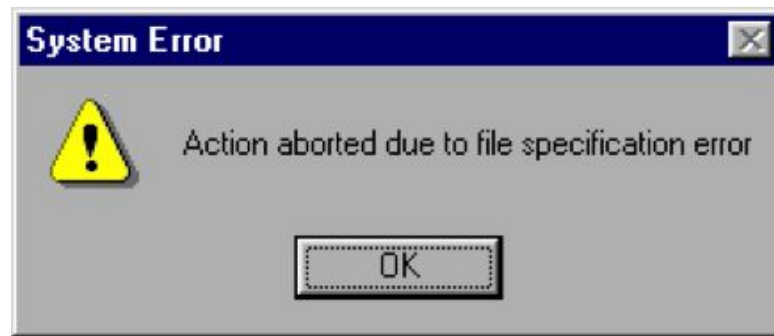


Tema 5

Testing con Jest

- **Introducción**
 - **Justificación y objetivos**
 - Tipos de pruebas
 - Calidad de las pruebas
 - Conclusiones

- Escribir **software libre de defectos**, es sumamente **difícil**
- La **especificación del comportamiento** es igualmente **desafiante**
- **No existen** métodos formales que se puedan aplicar a software real **para demostrar que no existen defectos**



- Una de las mejores formas que tienen los desarrolladores softwares de tener un **grado razonable de certeza** de que el software desarrollado se comporta como se espera es **probar su funcionamiento en ciertas circunstancias**
- A estas **ejecuciones o ensayos de funcionamiento** se las denomina "**pruebas**" o "**tests**"



Objetivos de las pruebas

- **Verificar** que el software funciona como fue diseñado
- **Evaluar las suposiciones** hechas en las especificaciones de requisitos y diseño a través de demostración **concreta**
- **Encontrar** errores en el software para que se puedan **corregir antes de que el software llegue a producción**

- **Nunca será posible garantizar la ausencia total de defectos de un producto en su paso a producción**
- Entre otras cosas, porque no se puede probar el software en sus (potencialmente) **infinitos usos**, sólo **un subconjunto de ellos**

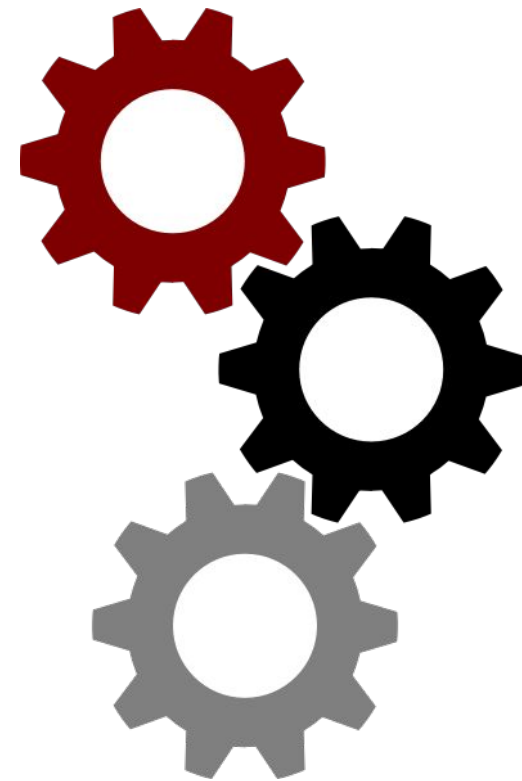
¿Qué es una prueba?

Las pruebas **comprueban** que un software se **comporta** como se **espera** en las **circunstancias** en las que se **prueba**, dentro de las **infinitas** formas de ser usado.

¿Qué es una prueba?

- **Ejecución del software**

- Las **pruebas** siempre implican la **ejecución** del programa en **ciertas circunstancias** (entradas, interacciones, estado de la base de datos...)
- Existen **otras actividades** dentro de la Verificación y Validación (V&V) como el **análisis de código fuente** que no necesitan ejecución, pero no se consideran pruebas.



¿Qué es una prueba?

- **Conjunto Finito de Casos de prueba**
 - Salvo raras excepciones, cualquier software puede usarse de **infinitas formas**.
 - Hacer pruebas implica un **equilibrio** entre **recursos** y **tiempo** limitados.
 - Es necesario seleccionar un **subconjunto finito** de **casos de prueba** dentro de las infinitas formas que existen de usar un software
 - Existen diversas técnicas para hacerlo

¿Qué es una prueba?

- **Comportamiento Esperado**

- Al realizar una prueba se debe determinar si el resultado observado es el esperado o no.
- Si no se conoce el resultado esperado, no tiene sentido realizar la prueba porque no se sabría distinguir entre un funcionamiento correcto y uno incorrecto.
- El comportamiento esperado puede ser comparado con
 - Las expectativas de los usuarios (**pruebas de aceptación**)
 - Una especificación de requisitos (**pruebas de verificación**)
 - Comportamiento esperado desde los **requisitos implícitos**.



Ejecución de un caso de prueba

- Al ejecutar un caso de prueba (de forma manual o automática) pueden ocurrir dos cosas:
 - Que la prueba sea un éxito (SUCCESS): El sistema se comporta de la forma esperada. Cumple con los requisitos. En esa prueba no se observa ningún defecto. (VERDE)
 - Que la prueba falle (FAIL): El sistema no se comporta de la forma esperada. No cumple con los requisitos. La prueba ha puesto de manifiesto un defecto o bug en el Software. (ROJO)



- **Importancia de las pruebas**

- Las pruebas se aceptan como una de las mejores formas de asegurar la calidad de un sistema
- Entre el **30% y 40% del tiempo** de un proyecto se puede dedicar a pruebas (Pressman, 2010)
- Cada **requisito, caso de uso o historia de usuario** debería **probarse** para asegurarse de que se ha implementado y se obtienen los resultados esperados.
- Las **pruebas no garantizan la ausencia de defectos**, pero ayudan a que no existan, al menos, en los escenarios para los que se ha **realizado una prueba**

- **Característica del software (*feature*):** funcionalidad o aspecto no funcional que ofrece un software y que se puede comprobar
- **Sujeto bajo Prueba (*Subject under Test* - SUT):** software que estamos probando. El SUT se define siempre desde la perspectiva de la prueba. A veces conocido como *Software under Test*.
- **Prueba o Caso de Prueba (*Test Case*):** Procedimiento para validar/verificar el SUT. Usualmente está formado por un estado inicial, una lista de pasos para interactuar con el SUT y el resultado esperado

- **Introducción**
 - Justificación y objetivos
 - **Tipos de pruebas**
 - Calidad de las pruebas
 - Conclusiones

- Existen **muchos tipos de pruebas**
 - Las pruebas se pueden clasificar atendiendo a **diferentes criterios** (tamaño, quién las crea, qué prueban...)
 - No hay una **taxonomía de pruebas globalmente aceptada**.
 - El **mismo nombre** se puede usar por **colectivos diferentes** para nombrar **diferentes tipos de pruebas**
 - Se presentarán los **tipos de pruebas más aceptados** indicando las ambigüedades cuando existan

- Existen muchas formas de clasificar las pruebas:
 - **Qué características prueban:** Funcionales o no funcionales
 - **Qué es el SUT:** Sistema, unidad, integración
 - **Cómo se ejecutan:** Manuales o automáticas
 - **Con qué objetivo se hace la prueba:** Aceptación, smoke (humo), sanity (sanidad)
 - **Con qué conocimientos se diseñan:** Caja blanca frente a Caja negra

- Existen muchas formas de clasificar las pruebas:
 - **Qué características prueban:** Funcionales o no funcionales
 - **Qué es el SUT:** Sistema, unidad, integración
 - **Cómo se ejecutan:** Manuales o automáticas
 - **Con qué objetivo se hace la prueba:** Aceptación, smoke (humo), sanity (sanidad)
 - **Con qué conocimientos se diseñan:** Caja blanca frente a Caja negra

Qué características prueban

- **Pruebas Funcionales:**

- Pruebas que verifican la funcionalidad ofrecida por el SUT.
- Comprueban que el SUT, partiendo de una situación determinada, cuando se interactúa con él ofrece los resultados esperados

- **Ejemplo de prueba funcional:**

- **Given:** En una tienda de comercio electrónico, si el usuario está en la página de un producto y tiene la cesta vacía
- **When:** El usuario pulsa el botón de añadir producto a la cesta
- **Then:** El icono de la cesta aparece un 1 producto



Qué características prueban

- **Pruebas No Funcionales:**
 - Verifican varios aspectos del comportamiento del sistema a menudo relacionados con las "-ilidades":
 - Pruebas de rendimiento
 - Pruebas de carga
 - Pruebas de estabilidad
 - Pruebas de estrés
 - Pruebas de usabilidad
 - Pruebas de seguridad
 - ...



- **Pruebas de Rendimiento:** verifican que el SUT cumple con el rendimiento esperado que se puede medir en función del número de transacciones por segundo, tiempo máximo de generación de respuesta, etc.
- **Prueba de carga o de esfuerzo o escalabilidad:** verifican que el SUT cumple con el comportamiento y rendimiento esperado bajo una pesada carga de los datos, la repetición de ciertas acciones de los datos de entrada, los grandes valores numéricos, consultas grandes a una base de datos o para comprobar el nivel de los usuarios concurrentes

- **Prueba de estabilidad:** es una prueba que se ejecuta durante un tiempo largo y que busca anomalías como pérdidas de memoria u otras degradaciones por la continuidad de la ejecución
- **Pruebas de estrés o volumen:** donde se escala la cantidad de carga con el tiempo hasta que se encuentren los límites del sistema; con el objetivo de examinar cómo falla y vuelve a su funcionamiento normal.

- **Herramientas**

- Específicas de cada tipo de test no funcional



Pruebas de rendimiento, carga
estrés y volumen de sistema

JMH

Java Microbenchmarking Harness

Pruebas de rendimiento unitarios
y de integración



**OWASP
ZAP**

Pruebas de seguridad de
aplicaciones web

- Existen muchas formas de clasificar las pruebas:
 - **Qué características prueban:** Funcionales o no funcionales
 - **Qué es el SUT:** Sistema, unidad, integración
 - **Cómo se ejecutan:** Manuales o automáticas
 - **Con qué objetivo se hace la prueba:** Aceptación, smoke (humo), sanity (sanidad)
 - **Con qué conocimientos se diseñan:** Caja blanca frente a Caja negra

- **Pruebas de sistema (eze)**
 - Las pruebas permiten verificar si el **software se comporta como se espera**.
 - Implícitamente estamos asumiendo que se **prueba el sistema completo**, que es sobre el que se definen los requisitos.
 - En estas pruebas el SUT (Sujeto bajo prueba) es el **sistema completo**
 - A veces se las llama **pruebas extremo a extremo, *end to end*** o **eze** (aunque existen ciertas diferencias)

Qué es el SUT

Enviar a Micael
Móstoles 28933

Todos los departamentos

Amazon.es de Micael

Ofertas

Cheques regalo

Vender

Ayuda

Hola Micael

Cuenta y listas

Pedidos

Mi Prime

Cesta

Libros en idiomas extranjeros

Búsqueda avanzada

Todos los géneros

Preventa

Los más vendidos

Todos los Libros

Catalán

Gallego

Euskera

Inglés

Compra hasta el 3 y recíbelo para Reyes con Envío 1 día GRATIS

Libros en idiomas extranjeros xunit testing software book

xUnit Test Patterns: Refactoring Test Code (Addison-Wesley Signature)

(Inglés) Tapa dura – jun 2007

de Gerard Meszaros (Autor)

★★★★☆ 31 opiniones de EE. UU.

Ver los 2 formatos y ediciones

Versión Kindle
EUR 35,37

Tapa dura
EUR 47,16 prime

Leer con nuestra App gratuita

3 Usado desde EUR 76,02

5 Nuevo desde EUR 47,16

Recíbelo antes de Reyes Magos.

¿Quieres recibirlo el viernes 29 dic.? Cómpralo antes de 22 hrs y 36 mins y elige Envío 1 día al completar tu pedido. Ver detalles

Automated testing is a cornerstone of agile development. An effective testing strategy will deliver new functionality more aggressively, accelerate user feedback, and improve quality. However, for many developers, creating effective automated tests is a unique and unfamiliar challenge.xUnit Test Patterns is the definitive guide to writing automated tests using xUnit, the most popular unit testing framework in use today. Agile coach and test automation expert Gerard Meszaros describes 68 proven patterns for

Leer más

Avisar de alguna información del producto errónea.

Echa un vistazo



Ver las 3 imágenes

Compartir

EUR 47,16 prime

Precio recomendado: EUR 55,05

Ahorras: EUR 7,89 (14%)

Precio final del producto

En stock.

Vendido y enviado por Amazon. Se puede envolver para regalo.

Cantidad: 1

Añadir a la cesta

Activar el pedido en 1-Clic

Enviar a Micael - Móstoles 28933

Añadir a la Lista de deseos

¿Tienes uno para vender?

Vender en Amazon

Comprados juntos habitualmente

- **Pruebas de sistema: Limitaciones**

- **Costosas de implementar:** probar un interfaz de usuario o una web simulando un usuario es complejo. Hay que tener en cuenta animaciones, selección de elementos en la interfaz, esperas, etc... La verificación de que el resultado es el esperado puede ser compleja (analizar un PDF...)
- **Costosas de ejecutar:** Los sistemas son cada vez más complejos y necesitan más elementos: Base de datos, servidor web, sistemas externos, navegador web, etc... y ejecutar las pruebas en estos sistemas consume tiempo y recursos computacionales

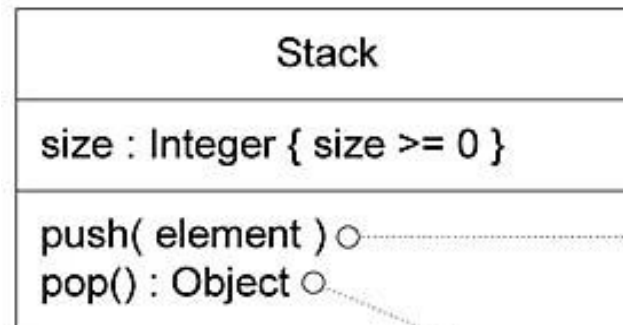
• Pruebas de sistema: Limitaciones

- **Frágiles:** Un cambio en la interfaz de usuario implica el cambio de todos los tests involucrados.
- **Poco flexibles:** Definir diferentes situaciones es complicado. Hay que generar conjuntos de datos con múltiples estados. Simular condiciones reales es complicado (por ejemplo: falta de conectividad con otros sistemas).

Para mitigar estos problemas, el
software puede probarse
por partes

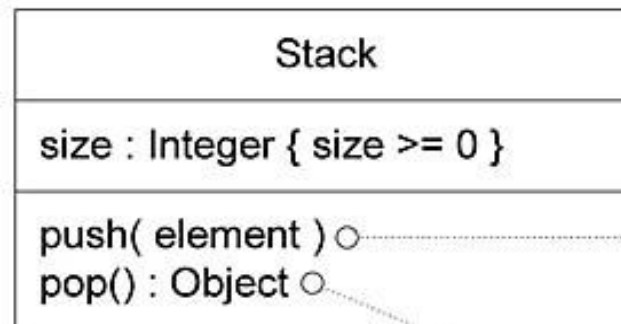
<https://martinfowler.com/bliki/TestPyramid.html>

Pruebas unitarias



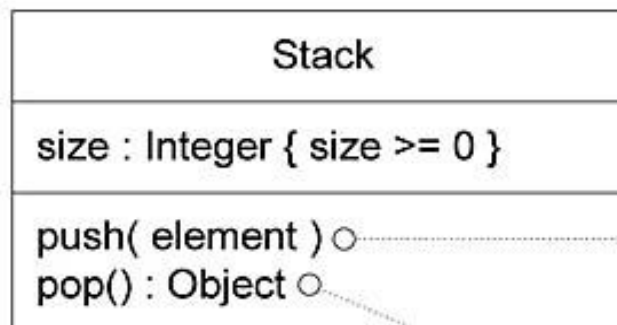
- **Pruebas unitarias**

- **Históricamente** las pruebas unitarias son aquellas en las que el SUT es un **elemento básico del software**: un método, una función, una clase, un módulo...



- **Pruebas unitarias**

- Se implementan en el mismo lenguaje que el SUT y se pueden hacer las llamadas con los parámetros que se necesiten y comprobar el resultado



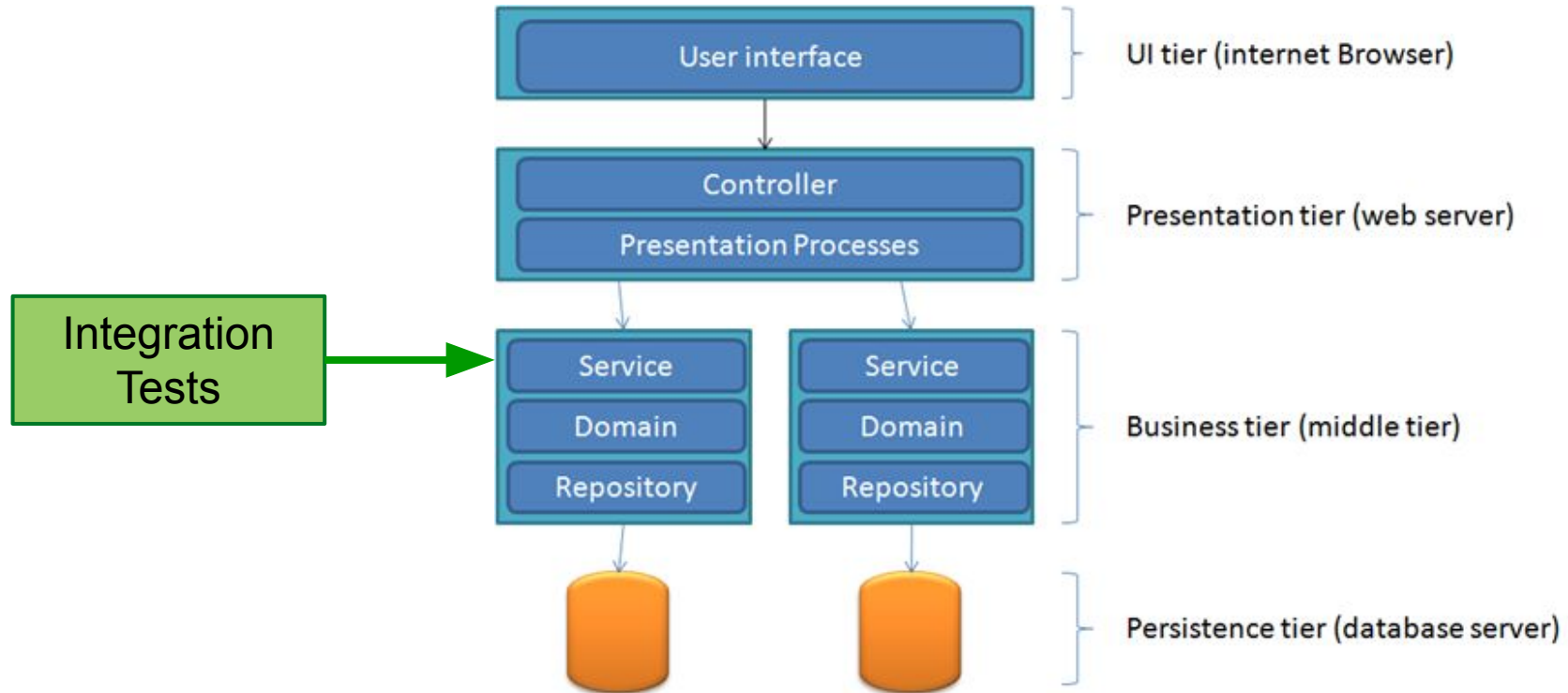
• Pruebas unitarias

- Nos focalizamos en probar una clase o método, consiguiendo **probar la mayoría de las situaciones posibles**
- El objetivo es que **su ejecución sea rápida** para que se puedan ejecutar **frecuentemente**
- Cuando se trabaja en una parte del código, las pruebas unitarias relacionadas con esa parte deberían poder **ejecutarse en menos de 10 segundos**

• Pruebas unitarias

- No utilizan sistemas externos (disco, red, complejos de configurar...)
- Eso permite que sean **rápidas** de ejecutar y sean muy **robustas**
- En algunas ocasiones, se sustituyen los sistemas externos por **sustitutos** o **dobles** que permiten diseñar **mejor el escenario** de prueba (y ejecutar más **rápido**)

Pruebas de integración



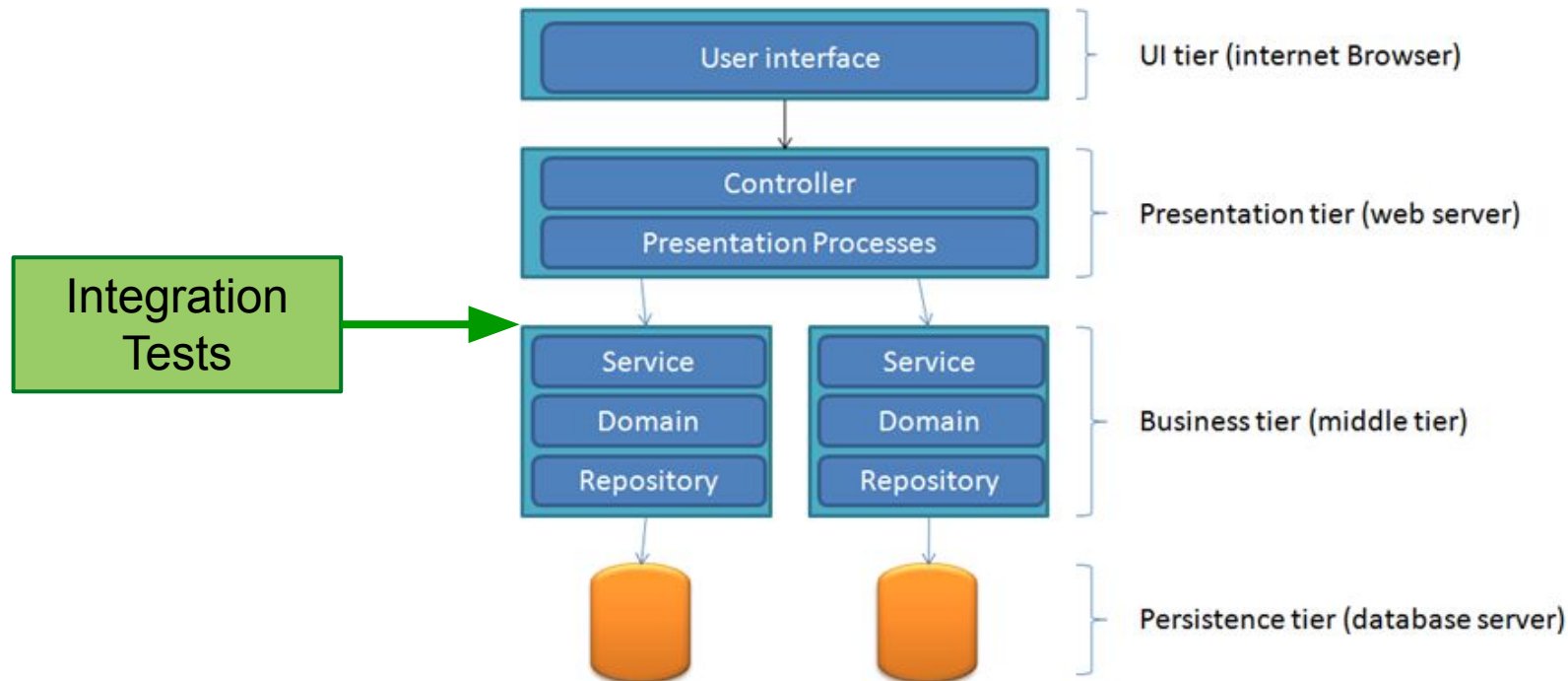
- **Pruebas de integración**

- Las pruebas de integración son aquellas en las que el SUT es un conjunto de **sub-sistemas de la aplicación interactuando entre sí**
- El objetivo de estas pruebas consiste en **verificar que la comunicación entre los sub-sistemas** sea correcta
- Habitualmente requiere la comunicación de diversos sistemas mediante **protocolos de red** o el uso de **librerías de terceros**

Qué es el SUT

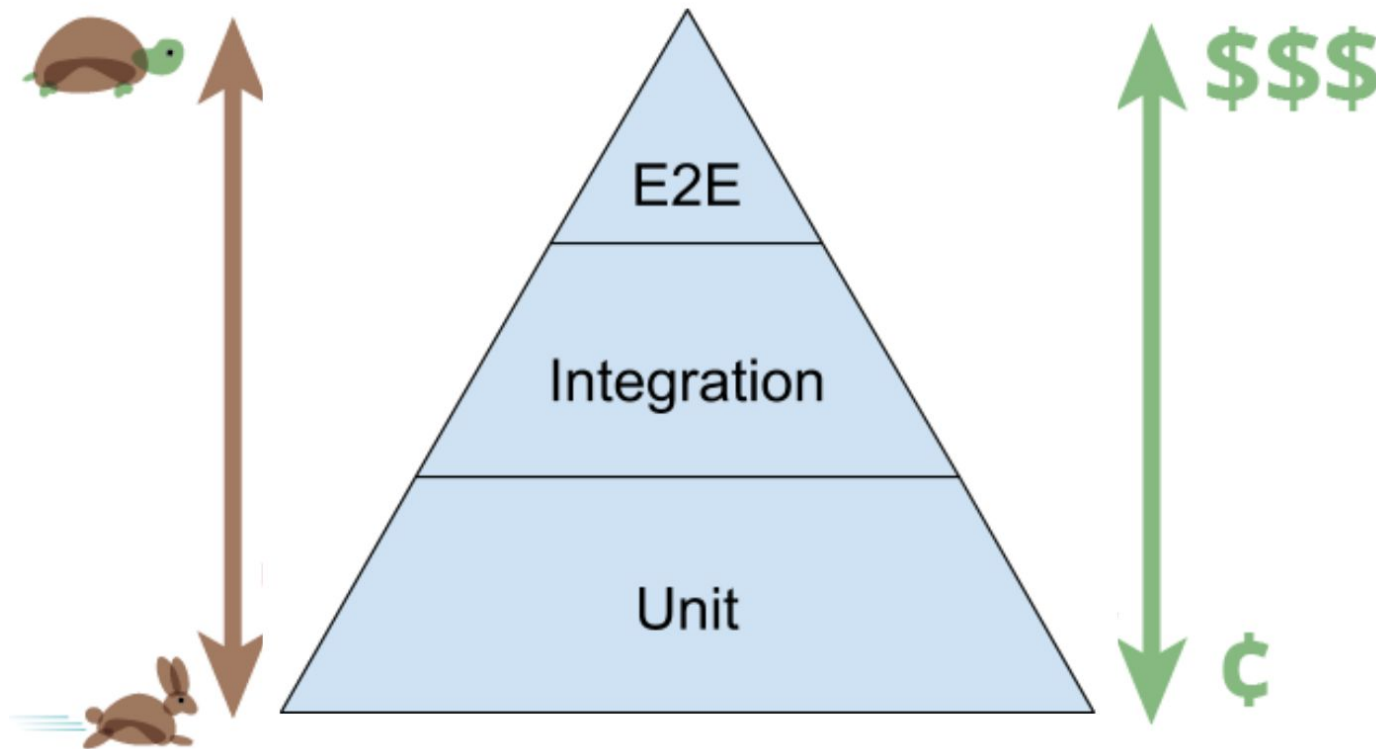
- Pruebas de integración

- Una prueba de integración puede ser probar la **lógica de negocio junto con la capa de acceso a base de datos** sin usar la interfaz de usuario



Qué es el SUT

- Pirámide de tests



<https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html>

- Google recomienda **70% Unitarios, 20% integración, 10% de sistema**

- Existen muchas formas de clasificar las pruebas:
 - **Qué características prueban:** Funcionales o no funcionales
 - **Qué es el SUT:** Sistema, unidad, integración
 - **Cómo se ejecutan:** Manuales o automáticas
 - **Con qué objetivo se hace la prueba:** Aceptación, smoke (humo), sanity (sanidad)
 - **Con qué conocimientos se diseñan:** Caja blanca frente a Caja negra

• Pruebas manuales

- La prueba que es **realizada por una persona** interactuando con el SUT.
- El usuario puede seguir algún **orden** o **guión** o con pruebas **exploratorias**.
- Pruebas exploratorias:
 - No tienen un guión específico.
 - El probador "explora" el sistema, con las hipótesis acerca de cómo debe comportarse en base a lo que en la aplicación ya se ha hecho y luego prueba esas hipótesis para ver si se sostienen.
 - Si bien no existe un plan rígido, las pruebas exploratorias son una actividad disciplinada con la que es más probable encontrar errores reales que con las pruebas de forma rígida con un guión.

Cómo se ejecutan

Test Case

gm-1:GmailLogin

Version 1

Summary
This Testcase is to test the login functionality of gmail Page.

Preconditions
1. User should be connected to the internet.
2. User should have valid gmail username and password.

#	Step actions	Expected Results	Execution		
1	Open Gmail Website	The Website should be opened.	Manual	✗	+
2	Enter username in the username textbox.	textbox should accept the entered data.	Manual	✗	+
3	Enter password in the password textbox.	textbox should accept the entered data.	Manual	✗	+
4	Click on "signin" button.	Login should success and navigate to the mail box page.	Manual	✗	+

Create step Resequence Steps

Status : Draft Importance : Medium Execution type : Manual Estimated exec. (min) : Save



- Herramientas para gestionar pruebas
 - Aplicación web que registra el plan de pruebas y los resultados de las ejecuciones tanto **manuales** como **automáticas**



- **Pruebas automáticas**

- Pruebas que para su ejecución requieren la ejecución de un **código de pruebas** (código escrito específicamente para probar el SUT).
- Es posible **verificar con mayor eficacia y eficiencia** el comportamiento del software porque se realiza de forma automática
- Dependiendo de lo que haya que probar, **automatizar una prueba puede ser muy costoso**: Tecnologías de interfaz de usuario complejas de probar de forma automática.

Cómo se ejecutan

- Pruebas automáticas

```
public class ListTests {  
  
    @Test  
    void addElementTest() {  
  
        List<String> list = new ArrayList<>();  
        list.add("Elem1");  
        assertEquals(list.size(), 1);  
    }  
}
```

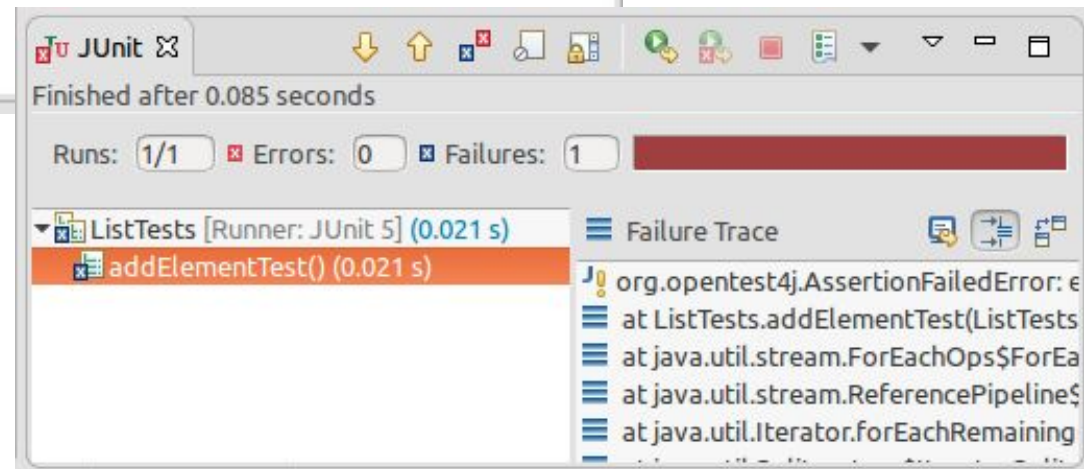
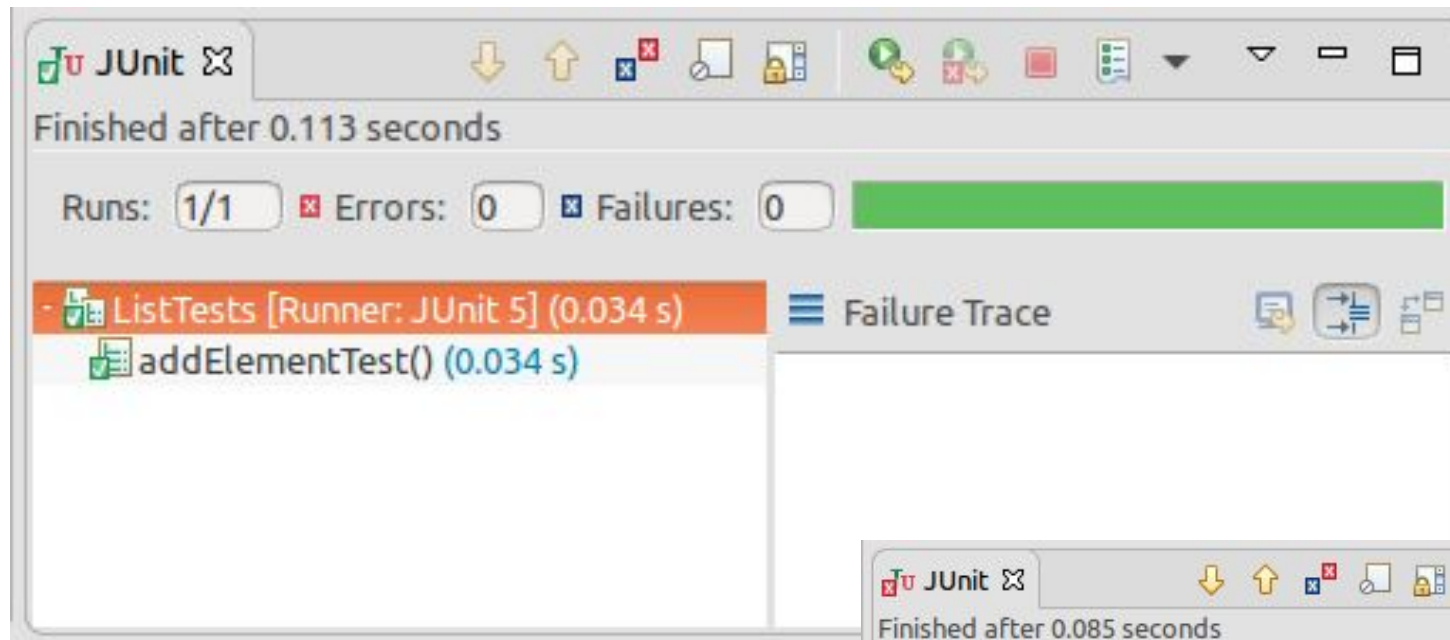
Given
(Situación inicial)

When
(Acciones sobre el
SUT)

Then
(Verificación del
resultado esperado)

Cómo se ejecutan

- Pruebas automáticas



- **Herramientas para automatizar pruebas**
 - Cómo se crea la prueba
 - Basadas en programación (*scripted*)
 - Basadas en grabación (*recorded*)
 - Cómo se interactúa con el SUT
 - Interfaz gráfica
 - Protocolo de red (REST, GraphQL, gRPC, JDBC)
 - API de programación

- **Herramientas para automatizar pruebas**
 - Basadas en programación (*scripted*):
 - El **desarrollador programa** el código de las pruebas manualmente.
 - El **lenguaje de programación de las pruebas puede ser un lenguaje de programación estándar** (Java, JavaScript, Python, C++...) o lenguajes específicos para testing
 - Hay que tener **conocimientos de programación**, pero los tests son **robustos** (el código que se puede adaptar, parametrizar, etc.)

- **Herramientas para automatizar pruebas**
 - Basadas en programación (*scripted*):
 - Se utilizan **frameworks y/o librerías** de programación para el ciclo de vida de los tests y verificar que el comportamiento obtenido es el esperado
 - Cuando se interactúa con el SUT mediante la **interfaz de usuario** o mediante **protocolos de red** se usan otras librerías específicas

- Frameworks y librerías para el ciclo de vida y las verificaciones

JUnit 



Hamcrest

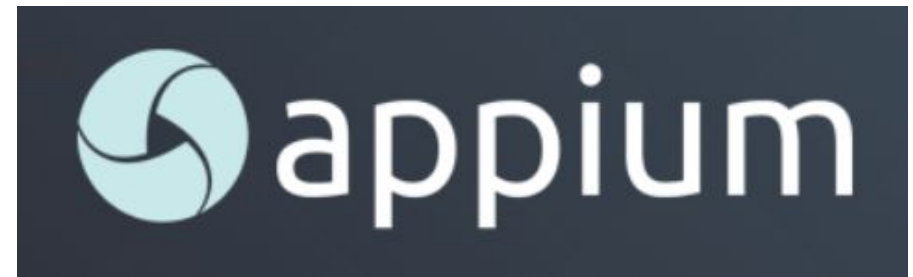
 **Jasmine**



 KARMA

mockito 

- Librerías para interactuar con el SUT



- **Herramientas para automatizar pruebas**
 - Basadas en grabación (*recorded*):
 - Un usuario prueba manualmente y las interacciones que realiza y los resultados obtenidos se graban.
 - Esa grabación se reproduce al ejecutar la prueba
 - Son más fáciles de usar, pero son más frágiles, cuando algún elemento del interfaz cambia hay que volver a grabar
 - Se utilizan mayormente en interfaz de usuario y protocolos

- Herramientas de grabación



Selenium IDE

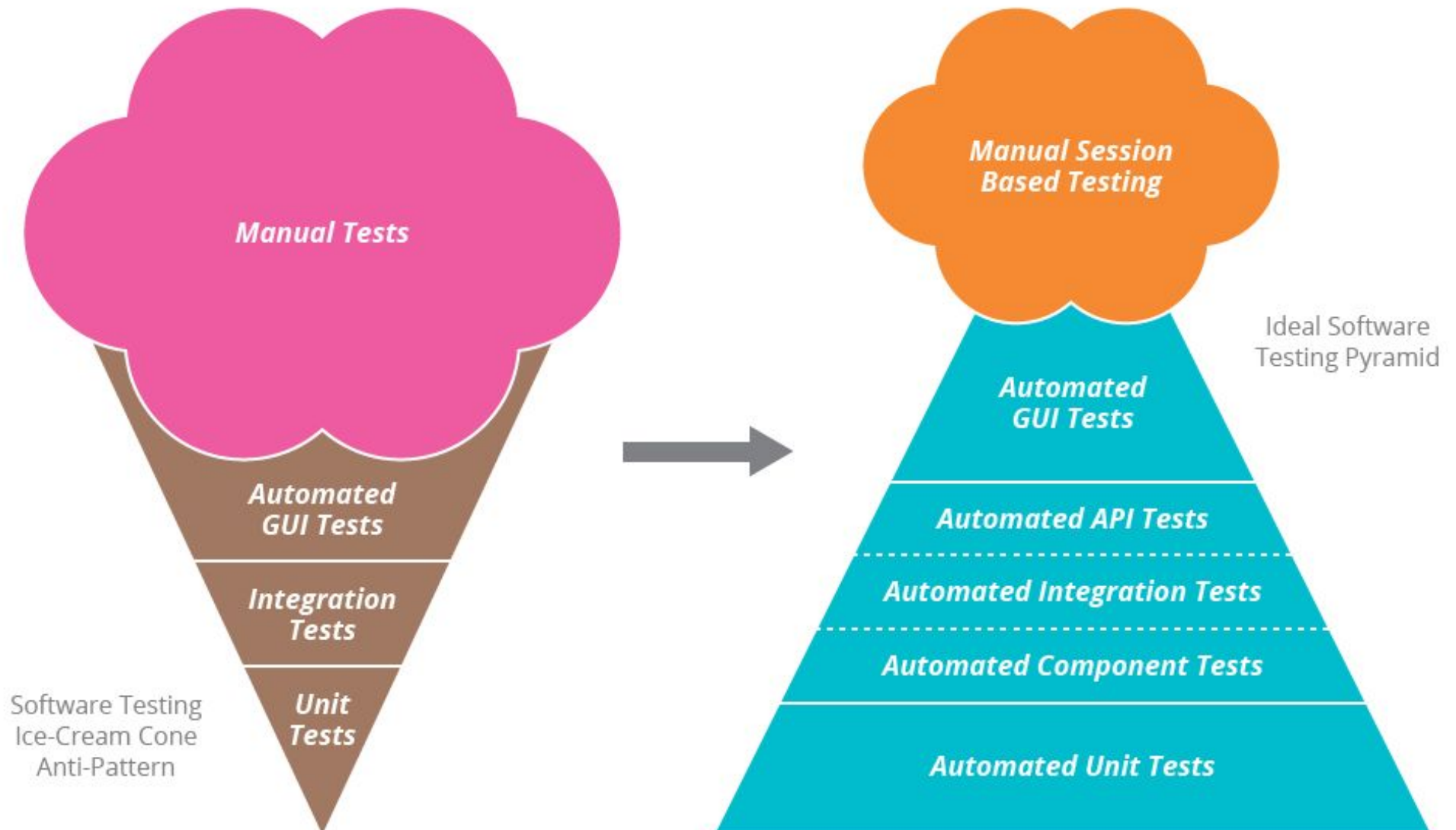


TestComplete

- **Pruebas manuales vs automáticas**

- Hay pruebas que pueden realizarse de cualquier forma:
 - Las pruebas de sistema, pruebas de integración cuando se usan protocolos de red para comunicación (API REST, Consultas a la BBDD...)
 - Es preferible que se automaticen para reducir el coste)
- **Otras pruebas sólo tienen sentido de forma automática:** Pruebas unitarias, pruebas de integración, pruebas de carga, pruebas de estrés...
- **Otras sólo tienen sentido de forma manual:** Pruebas de usabilidad, pruebas de sistema exploratorias...

Cómo se ejecutan



- Existen muchas formas de clasificar las pruebas:
 - **Qué características prueban:** Funcionales o no funcionales
 - **Qué es el SUT:** Sistema, unidad, integración
 - **Cómo se ejecutan:** Manuales o automáticas
 - **Con qué objetivo se hace la prueba:** Aceptación, smoke (humo), sanity (sanidad), regresión
 - **Con qué conocimientos se diseñan:** Caja blanca frente a Caja negra

Con qué objetivo se hace la prueba

- **Pruebas de aceptación:**

- Para el ISTQB (*International Software Testing Qualifications Board*):
 - **Pruebas de sistema** que permiten validar si el sistema cumple con los objetivos para los que fue diseñado.
 - Se **ejecutan manualmente por usuarios del sistema**
 - Por ejemplo **versiones beta** que prueban los usuarios
- Para la comunidad agile
 - Pruebas que definen las **reglas de negocio** que tiene que ofrecer la aplicación
 - Se ejecutan de forma **automática**
 - Pueden ser de **sistema**, de **integración** o incluso **unitarias**

https://en.wikipedia.org/wiki/Acceptance_testing

Con qué objetivo se hace la prueba

- **Prueba de humo (smoke test):**
 - Prueba de sistema que tiene como objetivo saber si el sistema está desplegado. Es previa a la ejecución de pruebas más exhaustivas. Para una web, saber si atiende peticiones http. Usadas después de un despliegue.
- **Pruebas de sanidad (sanity check):**
 - Prueba las funcionalidades básicas del sistema. Usadas después de un despliegue.

Con qué objetivo se hace la prueba

- **Prueba de regresión:**

- Son las pruebas que se ejecutan con el objetivo de determinar que las funcionalidades existentes no se han visto alteradas por nuevas funcionalidades incorporadas
- Todas las pruebas se convierten en pruebas de regresión cuando la funcionalidad es estable

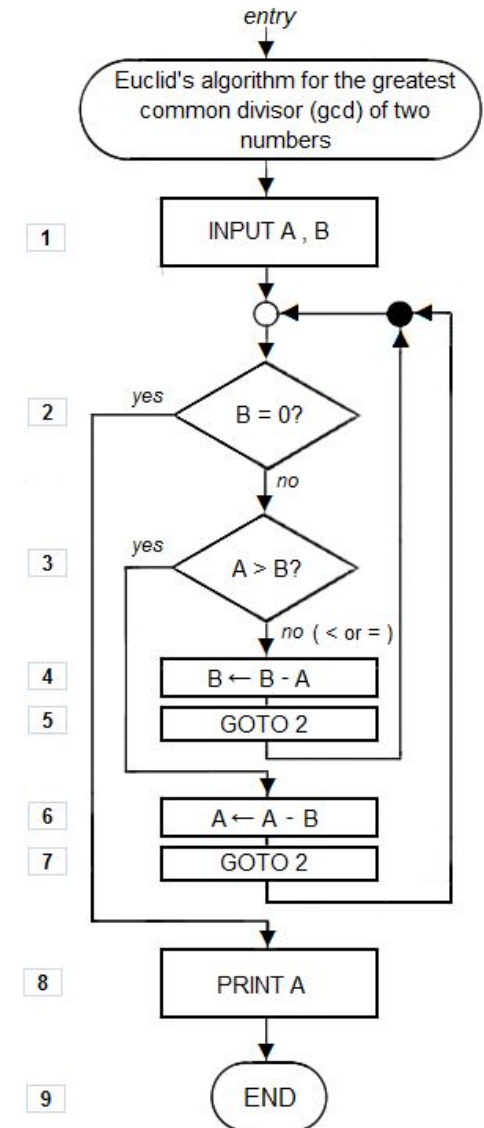
- Existen muchas formas de clasificar las pruebas:
 - **Qué características prueban:** Funcionales o no funcionales
 - **Qué es el SUT:** Unidad, componente, integración, sistema
 - **Cómo se ejecutan:** Manuales o automáticas
 - **Con qué objetivo se hace la prueba:** Aceptación, smoke (humo), sanity (sanidad)
 - **Con qué conocimientos se diseñan:** Caja blanca frente a Caja negra

- **Caja Negra (Black-box):**
 - Sólo se tiene en cuenta la descripción de la **funcionalidad observable del SUT**.
 - Verifican si se obtiene la **salida deseada** a una **entrada dada**.
 - Se utiliza generalmente en pruebas de **sistema e integración**
 - Idealmente por **equipos no involucrados en el desarrollo** (para no estar contaminados por detalles de implementación o asunciones en la interpretación de requisitos)

Con qué conocimientos se diseñan

- **Caja blanca (White-box):**

- Se tiene en cuenta cómo está construido el **SUT internamente**
- Permite que todo el **código interno sea ejercitado** durante las pruebas. Todos los caminos independientes son ejecutados
- Tienen que realizarse por los **desarrolladores** que han implementado el SUT.



- **Introducción**
 - Justificación y objetivos
 - Tipos de pruebas
 - **Calidad de las pruebas**
 - Conclusiones

- Un **software se puede usar de formas infinitas**, pero no podemos diseñar y ejecutar infinitas pruebas
- **Cuántas pruebas** hay que implementar?
- Cuándo el software se puede considerar **suficientemente probado**?
- Existen **técnicas** para determinar la **calidad de las pruebas**

- **Cobertura de código (*code coverage*)**
 - Es el % de código del SUT ejecutado cuando se ejecuta el conjunto de pruebas
 - Cobertura del 100%
 - No implica que el SUT no tenga defectos
 - Una misma línea de código puede ejecutarse correctamente con unos valores de variables e incorrectamente con otros.
 - A veces es muy complicado llegar a una cobertura del 100% porque hay código genérico que sólo se ejecuta en situaciones difíciles de probar
 - Se considera como valor aceptable una cobertura 70-80%

- Cobertura de código (*code coverage*)

JaCoCo

```
public boolean addAll(int index, Collection c) {  
    if(c.isEmpty()) {  
        return false;  
    } else if(_size == index || _size == 0) {  
        return addAll(c);  
    } else {  
        Listable succ = getListableAt(index);  
        Listable pred = (null == succ) ? null : succ.prev();  
        Iterator it = c.iterator();  
        while(it.hasNext()) {  
            pred = insertListable(pred, succ, it.next());  
        }  
        return true;  
    }  
}
```

1 of 2 branches missed.
Press 'F2' for focus

<http://www.eclemma.org/>

build

passing

coverage

- **Mutaciones (*Mutation testing*)**

- El código del SUT se modifica de forma automática (mutación) creando un nuevo SUT (mutante)
- Se ejecutan los tests contra el SUT mutante

Si los tests fallan

- El mutante muere.
- Los tests son de calidad porque han detectado el cambio de comportamiento.

Si los tests no fallan

- El mutante vive.
- Los tests son mejorables porque no han sido capaces de detectar el cambio de comportamiento

- **Mutaciones (*Mutation testing*)**



pitest.org

Mutation Testing para Java



Muy costosa de ejecutar. No está muy extendida

- **Introducción**
 - Justificación y objetivos
 - Tipos de pruebas
 - Calidad de las pruebas
 - **Conclusiones**

- Las **pruebas automáticas** son una de las técnicas más usadas para **crear código con pocos defectos**:
 - Junto con la **integración continua** se **verifica** de forma continua que el software se **comporta como se espera**
 - **Evitan regresiones** durante el ciclo de desarrollo
 - Ayudan en el **desarrollo** y a la **comunicación con negocio**
- Las **pruebas manuales** tienen su utilidad en pruebas de sistema **exploratorias** y en pruebas de **aceptación**

- Implementar pruebas automáticas **no es sencillo**
 - Existen muchos **tipos de pruebas** (algunas más complejas de implementar que otras)
 - Existen **muchas librerías y herramientas de apoyo** para la implementación y ejecución de todos tipos de pruebas (síntoma de que no es sencillo)
 - Cada **lenguaje de programación / plataforma** tiene sus propias herramientas y librerías

- Existen formas de medir la **calidad de las pruebas automáticas**
 - Cobertura del código (*Code coverage*)
 - Pruebas de mutación (*Mutation testing*)
- **Coste** de las pruebas automáticas
 - Implementar pruebas automáticas tiene asociado un coste de implementación y mantenimiento.
 - Hay que llegar a un compromiso de valor aportado (defectos que no llegan a producción) frente a coste.

- Introducción
- **Pruebas unitarias**
 - Introducción
 - Casos de Test
 - Aserciones
 - Tests asíncronos
 - Dobles
 - Conclusiones

- Cada **lenguaje de programación** dispone de uno o varios **frameworks** para implementar pruebas automáticas unitarias y de integración
- Los **frameworks** aprovechan toda la potencia del lenguaje para facilitar la tarea del desarrollador de tests

- Aunque existen algunas diferencias dependiendo del lenguaje, **todos los frameworks son muy parecidos:**
 - Cada **test** se implementa en una **función / método**
 - Existen **funciones / métodos** para las **aserciones**

• Test en Java con el framework JUnit

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

public class MyTests {

    @Test
    public void multiplicationOfZeroIntegersShouldReturnZero() {

        // MyClass is tested
        MyClass tester = new MyClass();

        // assert statements
        assertEquals("10 x 0 must be 0", 0, tester.multiply(10, 0));
        assertEquals("0 x 10 must be 0", 0, tester.multiply(0, 10));
        assertEquals("0 x 0 must be 0", 0, tester.multiply(0, 0));
    }
}
```

• Test en Groovy con Spock

```
class InterpolateServiceTest extends Specification {  
    @Shared def interpolateService = new InterpolateService()  
  
    def "interpolate two numbers with even no. of steps"() {  
        expect:  
            interpolateService.interpolate(a, b, c) == d  
  
        where:  
            a | b | c | d  
            5.0 | 25.0 | 4 | [5.0, 10.0, 15.0, 20.0, 25.0]  
            2.0 | 14.0 | 6 | [2.0, 4.0, 6.0, 8.0, 10.0, 12.0, 14.0]  
    }  
}
```

- **Test en JavaScript con Jasmine**

```
describe( "distance converter", function () {  
    it("converts inches to centimeters", function () {  
        expect(Convert(12, "in").to("cm")).toEqual(30.48);  
    });  
    it("converts centimeters to yards", function () {  
        expect(Convert(2000, "cm").to("yards")).toEqual(21.87);  
    });  
});
```

- **Test en C++ con GoogleTest**

```
// Tests factorial of 0.
TEST(FactorialTest, HandlesZeroInput) {
    EXPECT_EQ(1, Factorial(0));
}

// Tests factorial of positive numbers.
TEST(FactorialTest, HandlesPositiveInput) {
    EXPECT_EQ(1, Factorial(1));
    EXPECT_EQ(2, Factorial(2));
    EXPECT_EQ(40320, Factorial(8));
}
```

- Vamos a estudiar la librería **Jest** para testing JavaScript



<https://jestjs.io/>

Testing JavaScript con Jest

- Instalación en un proyecto NPM

```
$ npm install --save-dev jest
```

- Añadir como script de test en package.json

```
{  
  "scripts": {  
    "test": "jest"  
  }  
}
```

Testing JavaScript con Jest

ejem1

- Mi primer test

sum.js

```
function sum(a, b) {  
  return a + b;  
}  
module.exports = sum;
```

sum.test.js

```
const sum = require('./sum');  
  
test('adds 1 + 2 to equal 3', () => {  
  expect(sum(1, 2)).toBe(3);  
});
```

Testing JavaScript con Jest

- Ejecución de los tests

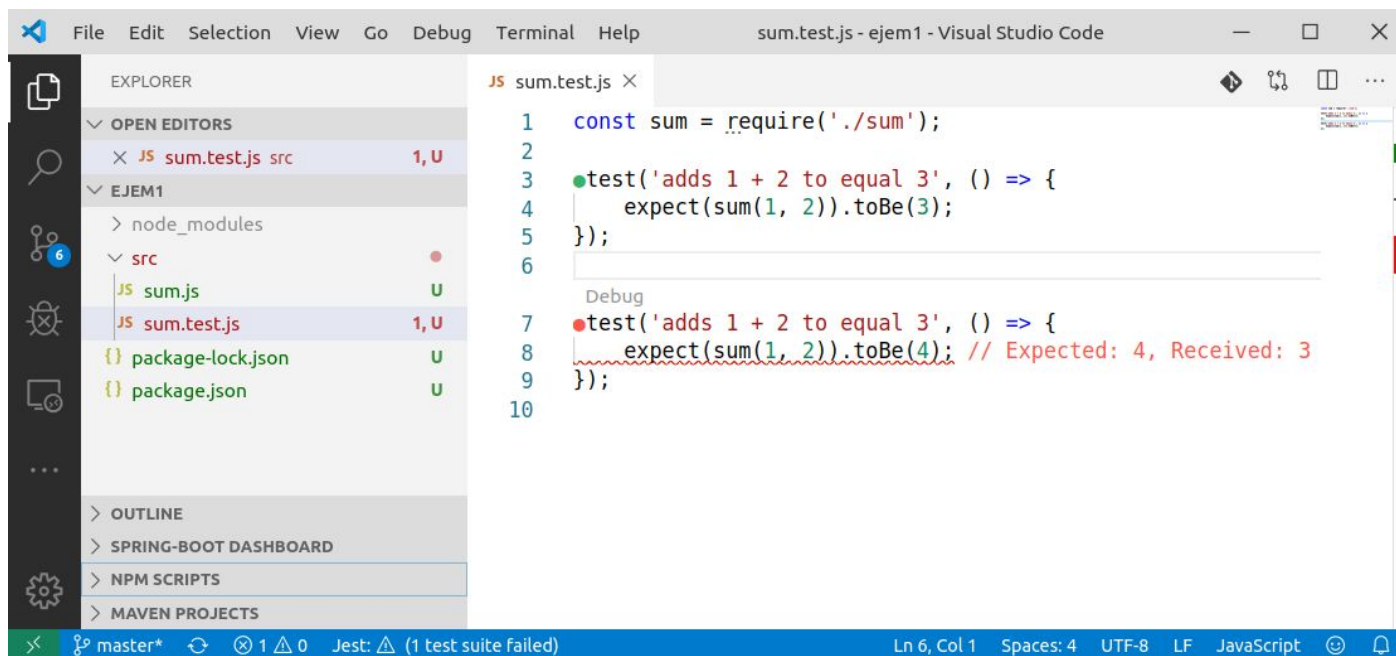
```
$ npm test
```

```
PASS src/sum.test.js
  ✓ adds 1 + 2 to equal 3 (2ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
```


Jest en Visual Studio Code

- Extensión “jest” (Orta)
- El package.json debe estar en la raíz
- Los tests se ejecutan automáticamente



The screenshot shows the Visual Studio Code interface with the Jest extension installed. The Explorer sidebar on the left shows the project structure with files like `sum.test.js`, `sum.js`, `package-lock.json`, and `package.json`. The main editor displays the content of `sum.test.js`, which contains two Jest tests. The first test, `test('adds 1 + 2 to equal 3', () => { expect(sum(1, 2)).toBe(3); });`, is marked with a green dot, indicating it passed. The second test, `test('adds 1 + 2 to equal 3', () => { expect(sum(1, 2)).toBe(4); // Expected: 4, Received: 3 });`, is marked with a red dot, indicating it failed. The status bar at the bottom shows the Jest test results: `Jest: 1 test suite failed`.

```
1 const sum = require('./sum');
2
3 test('adds 1 + 2 to equal 3', () => {
4   expect(sum(1, 2)).toBe(3);
5 });
6
7 test('adds 1 + 2 to equal 3', () => {
8   expect(sum(1, 2)).toBe(4); // Expected: 4, Received: 3
9 });
10
```

<https://github.com/jest-community/vscode-jest>

Configuración de Jest

- Se pueden configurar aspectos globales de jest
- Fichero jest.config.js

jest.config.js

```
module.exports = {  
  option: value,  
  option2: value,  
  ...  
}
```

- Se crea con

```
$ jest --init
```

<https://jestjs.io/docs/en/configuration.html>

- Introducción
- **Pruebas unitarias**
 - Introducción
 - **Casos de Test**
 - Aserciones
 - Tests asíncronos
 - Dobles
 - Conclusiones

- Los tests son llamadas a la función **test(...)**
- No hay que importarla (de eso se encarga el comando que ejecuta los tests)

sum.test.js

```
test('adds 2 + 2 to equal 4', () => {  
  expect(2 + 2).toBe(4);  
});
```

- Existen **3 partes** diferenciadas en la implementación de un test:
 - **Arrange / given:** Definir el estado inicial del SUT. Las condiciones del test
 - **Act / when:** Actuar sobre esa SUT. Ejercitarlo.
 - **Assert / then:** Verificar que el comportamiento obtenido es el esperado

Casos de Test

ejem1

calculadora.js

```
const Calculadora = require('./calculadora');

test('testSuma', () => {

  // Arrange / Given
  const calculadora = new Calculadora();

  // Act / When
  let resultado = calculadora.suma(1, 1);

  // Assert / Then
  expect(resultado).toBe(2);

});
```

- La verificación se realiza mediante **aserciones** (assertions) que determinan si una condición se cumple
- El **resultado** de una aserción puede ser:
 - **Éxito (success)**
 - **Fallo (fail)**
- Si la aserción es un **éxito**, el test **pasa**, si no, el test **falla**
- Un test puede tener **varias aserciones** (incluso podría no contener ninguna, pero no es una buena práctica)

- Un módulo de test puede tener varias funciones ejem1

sum.test.js

```
test('adds 2 + 2 to equal 4', () => {  
  expect(2 + 2).toBe(4);  
});  
  
test('adds 1 + 1 to equal 2', () => {  
  expect(1 + 1).toBe(2);  
});
```


- Aserciones sobre veracidad

```
test('null', () => {  
  const n = null;  
  expect(n).toBeNull();  
  expect(n).toBeDefined();  
  expect(n).not.toBeUndefined();  
  expect(n).not.toBeTruthy();  
  expect(n).toBeFalsy();  
});
```

<https://jestjs.io/docs/en/expect>

- Aserciones sobre veracidad

```
test('zero', () => {  
  const z = 0;  
  expect(z).not.toBeNull();  
  expect(z).toBeDefined();  
  expect(z).not.toBeUndefined();  
  expect(z).not.toBeTruthy();  
  expect(z).toBeFalsy();  
});
```

<https://jestjs.io/docs/en/expect>

- Aserciones sobre números

```
test('two plus two', () => {  
  const value = 2 + 2;  
  expect(value).toBeGreaterThan(3);  
  expect(value).toBeGreaterThanOrEqual(3.5);  
  expect(value).toBeLessThan(5);  
  expect(value).toBeLessThanOrEqual(4.5);  
  
  // toBe and toEqual are equivalent for numbers  
  expect(value).toBe(4);  
  expect(value).toEqual(4);  
});
```

<https://jestjs.io/docs/en/expect>

- Aserciones sobre números

```
test('adding floating point numbers', () => {  
  const value = 0.1 + 0.2;  
  
  //This won't work because of rounding error  
  // 0.1 + 0.2 is 0.30000000000000004  
  //expect(value).toBe(0.3);  
  
  //This works.  
  expect(value).toBeCloseTo(0.3);  
  // You can specify precision digits  
  expect(value).toBeCloseTo(0.3,5);  
});
```

<https://jestjs.io/docs/en/expect>

• Aserciones

<code>expect(...).toBe(expected)</code>	Se espera que sea igual (Con Object.is) a “expected”
<code>expect(...).toEqual(expected)</code>	Se espera que sea igual (comparación profunda propiedad a propiedad) a “expected”
<code>expect(...).toBeFalsy()</code>	Se espera que sea “falsy” (como en el if)
<code>expect(...).toBeTruthy()</code>	Se espera que sea “truthy” (como en el if)
<code>expect(...).toBeCloseTo(expected, digits?)</code>	Se espera que el valor sea igual al número decimal (hasta un nivel de precisión)
<code>expect(...).toBeGreaterThan(number)</code>	Se espera que el valor sea mayor o igual
<code>expect(...).toBeNull()</code>	Se espera que sea null
<code>expect(...).toContain(item)</code>	Se espera que el array contenga exactamente ese item
<code>expect(...).toContainEqual(item)</code>	Se espera que el array contenga un item igual
...	...

<https://jestjs.io/docs/en/expect>

- **Aserciones**

- Se pueden negar las aserciones con `.not`

```
test('the best flavor is not coconut', () => {  
  expect(bestFlavor()).not.toBe('coconut');  
});
```

<https://jestjs.io/docs/en/expect>

- En general es recomendable usar **expect(...)** para verificar que el resultado obtenido es el esperado
- A veces no es posible y hay que detectar en código si el SUT se comporta como se **espera**
- Se puede **forzar el fallo** del test con una excepción

```
it('force fail', () => {  
    //...  
    if (//condition//) {  
        throw new Error("Behavior not expected");  
    }  
});
```

• Ejercicio 1

- Implementa varios tests de la clase Complex (proporcionada)
- Comprueba que el complejo `Complex(0,0)` tiene la parte real y la parte imaginaria con valor 0
- Comprueba que `Complex(0,0)` es el valor neutro de la operación suma:

`Complex(0,0) + Complex(1,1) == Complex(1,1)`


`Complex(1,1) + Complex(0,0) == Complex(1,1)`

- **Test fixtures (Partes fijas de un test)**
 - Cuando hay varios tests que repiten la inicialización o la finalización de recursos se pueden mover esas partes comunes a funciones independientes
 - **beforeEach(...)** para definir código que se ejecutará antes de cada test
 - **afterEach(...)** para definir código que se ejecutará al finalizar cada test (aunque haya fallado)

Casos de Test

ejem1

calculadora2.test.js



```
const Calculadora = require('./calculadora');

// Arrange / Given
let calculadora;

beforeEach(() => {
  calculadora = new Calculadora();
});

test('testSuma', () => {

  // Act / When
  let resultado = calculadora.suma(1, 1);

  // Assert / Then
  expect(resultado).toBe(2);

});

test('testResta', () => { ... });
```

• Ejercicio 2

- Transforma el Ejercicio 1 para usar Test fixtures
- Define un atributo zero que se inicializa en un método setUp anotado como @Before
- Ese atributo se usará siempre que se necesite el número complejo $0+0i$

`zero + Complex(1,1) == Complex(1,1)`

`Complex(1,1) + zero == Complex(1,1)`

- **Test fixtures (Partes fijas de un test)**
 - A veces es conveniente ejecutar un código antes de todos los tests de un módulo y después de todos ellos
 - Las funciones **beforeAll(...)** y **afterAll(...)** se utilizan para ello

```
beforeAll(() => {  
  return initializeCityDatabase();  
});  
  
afterAll(() => {  
  return clearCityDatabase();  
});  
  
test('city database has Vienna', () => {  
  expect(isCity('Vienna')).toBeTruthy();  
});  
  
test('city database has San Juan', () => {  
  expect(isCity('San Juan')).toBeTruthy();  
});
```

- **Tests parametrizados**
 - En ciertas ocasiones es útil ejecutar varios tests que ejecutan la misma lógica pero sólo cambian en los datos utilizados
 - Se escribir un único test que probara con todos los valores (con un for sobre un array), pero un fallo en un valor pararía la ejecución del test

• Tests parametrizados

- Los tests parametrizados permiten definir un conjunto de datos con los que se ejecutará el test.
- Se crearán tantos tests como juegos de datos existan
- Si el test falla con algunos datos en concreto, se mostrará como un único test fallido informando de esos datos

```
test.each(data)('test name', (p1, p2...) => {  
  //test code  
});
```

<https://jestjs.io/docs/en/api#testeachtablename-fn-timeout>

```
const cases = [  
  [1, 1, 2],  
  [1, 2, 4],  
  [2, 1, 3],  
]  
  
test.each(cases)('add (%i, %i) = %i', (a, b, expected) => {  
  expect(a + b).toBe(expected);  
});
```


Casos de Test

ejem2

```
const cases = [  
  [1, 1, 2],  
  [1, 2, 4],  
  [2, 1, 3],  
]  
  
test.each(cases)('add (%i, %i) = %i', (a, b, expected) => {  
  expect(a + b).toBe(expected);  
});
```

Los datos se definen como un array de arrays.
Se ejecuta el test por cada "fila"

El nombre del test se puede parametrizar para incluir los valores de los parámetros (sintaxis `util.format(..)`)

Cada vez que se ejecuta el test se le pasan como parámetros los datos

Casos de Test

ejem2

```
FAIL src/each.test.js
  ✓ add (1, 1) = 2 (2ms)
  ✗ add (1, 2) = 4 (3ms)
  ✓ add (2, 1) = 3

  ● add (1, 2) = 4

    expect(received).toBe(expected) // Object.is equality

    Expected: 4
    Received: 3

       4 |         [2, 1, 3],
       5 |     ])(`add (%i, %i) = %i`, (a, b, expected) => {
    >  6 |         expect(a + b).toBe(expected);
         |                         ^
       7 |     });

    at src/each.test.js:6:19

Test Suites: 1 failed, 1 total
Tests:       1 failed, 2 passed, 3 total
Snapshots:  0 total
Time:        1.069s
Ran all test suites.
npm ERR! Test failed.  See above for more details.
```

- **Ejercicio 3**

- Implementa un test que verifique que el valor absoluto de un número complejo (**método abs()**) se calcula correctamente
- Define **varios número complejos** de ejemplo y verifica el cálculo para ellos
- Usa tests parametrizados

- Introducción
- **Pruebas unitarias**
 - Introducción
 - Casos de Test
 - **Aserciones**
 - Tests asíncronos
 - Dobles
 - Conclusiones

- Uno de los objetivos de las librerías de testing es que los **tests** sean **fáciles de escribir y de leer**
- Para ello, las **aserciones** no tienen que ser complejas
- La mayor parte de la innovación en librerías de testing ha ido en la **facilidad de escribir aserciones**, especialmente las específicas de un dominio concreto (**estructuras de datos, JSON, BBDD...**)

- Verificar que una **excepción** se lanza cuando debería hacerlo
- Los tests también tienen que verificar que el software **genera los errores esperados**
- El test debería fallar si no se genera la excepción o es de un tipo diferente al esperado

- **expect(..).toThrow()**

```
test('Exception Error should be thrown', () => {  
  expect(() => {  
    doSomething();  
  }).toThrow();  
});
```

- **expect(..).toThrowError(...)**

```
test('Exception Error with "error_message" should be thrown', () => {  
  expect(() => {  
    doSomething();  
  }).toThrowError('error_message');  
});
```

Se puede indicar que el mensaje debería contener un substring o el tipo de error esperado

• Ejercicio 4

- Implementar un test que verifique que el recíproco de Zero Complex(0,0) eleva una excepción con el mensaje "division by zero"
- Como ahora mismo la clase Complex no está implementada de esa forma, el test debería fallar
- Modifica la clase Complex para que realmente eleve una excepción cuando se intente calcular el recíproco de Complex(0,0)
- Después de modificar el SUT, el test debería pasar

- **Mejorar los mensajes de error**
 - Cuanto más claros y más información aporten los mensajes en caso de fallo del test, más rápido se solucionará el bug que lo origina
 - Existen diversas formas de mejorar los mensajes:
 - Usar las aserciones más concretas
 - Usar “*matchers*”

- **Usar las aserciones más concretas**
 - Se puede usar una aserción general **`expect(expression).toBeTruthy();`** que debería evitarse si existe una aserción más específica
 - Si se usa **`toBeTruthy`**, en caso de fallo, Jest sólo podrá mostrar la expresión y el resultado de la misma, pero no los valores individuales

Mejorar los mensajes de error

- Usar las aserciones más concretas

```
expect(getName() === 'juan').toBeTruthy();
```

● Expect name is Juan

```
expect(received).toBeTruthy()
```

Received: false

```
5 | test('Expect name is Juan', () => {  
6 |  
> 7 |     expect(getName() === 'juan').toBeTruthy();  
8 |  
9 | });  
10 |
```

at Object.<anonymous> (src/exception.test.js:7:34)

Sólo muestra que la
aserción no es
truthy

Mejorar los mensajes de error

- Usar las aserciones más concretas

`expect(getName()).toBe('juan');`

● Expect name is Juan with matcher

`expect(received).toBe(expected) // Object.is equality`

Expected: "juan"

Received: "pepe"

```
11 | test('Expect name is Juan with matcher', () => {
12 |
> 13 |     expect(getName()).toBe('juan');
    |                               ^
14 |
15 | });
16 |
```

at Object.<anonymous> (src/exception.test.js:13:23)

Muestra el valor de
cada operando

- Usar “*matchers*”

- Un *matcher* es una función que verifica si un valor cumple una determinada propiedad:
 - Es un String y cumple una expresión regular
 - Es una lista y contiene algunos elementos
 - Es una lista vacía
 - Es igual que otro valor...
- Además de mejorar el mensaje, mejora la legibilidad del test (similar a lenguaje natural)

- **Jest tiene varios tipos de matchers**
 - Como métodos **expect(...).toXX()**

```
expect(getName()).toBe('juan');
```

- Asymmetric matchers: Como valores pasados a **expect(...).toEqual(...)**

```
expect(array).toEqual(expect.arrayContaining(expected));
```

- Matchers definidos por el programador

- **Matchers expect(...).toXX():**
 - Strings (expresiones regulares)


```
test('there is no I in team', () => {  
  expect('team').not.toMatch(/I/);  
});  
  
test('but there is a "stop" in Christoph', () => {  
  expect('Christoph').toMatch(/stop/);  
});
```


- **Matchers expect(...).toXX():**
 - Estructuras de datos

```
const shoppingList = [  
  'diapers',  
  'kleenex',  
  'paper towels',  
  'beer',  
];  
  
test('the shopping list has beer on it', () => {  
  expect(shoppingList).toContain('beer');  
  expect(new Set(shoppingList)).toContain('beer');  
});
```

- Matchers expect(...).toXX():
 - Estructuras de datos

```
describe('my beverage', () => {  
  test('is delicious and not sour', () => {  
  
    const myBeverage = {delicious: true, sour: false};  
  
    expect(myBeverages()).toContainEqual(myBeverage);  
  
  });  
});
```



Para buscar objetos
(iguales) aunque no
sean "el mismo"

- **Asymmetric matchers**
 - Verificar el tipo: `any(...)`

3.test.js

```
test('Is a string', () => {
  expect(someString()).toEqual(expect.any(String));
});

test('Is a number', () => {
  expect(someNumber()).toEqual(expect.any(Number));
});

test('Is a Function', () => {
  expect(someFunction()).toEqual(expect.any(Function));
});

test('Is an Array', () => {
  expect(someArray()).toEqual(expect.any(Array));
});

test('Is an Object', () => {
  expect(someObject()).toEqual(expect.any(Object));
});
```

- **Asymmetric matchers**
 - Strings: `stringContaining(...)`

2.test.js

```
test('there is no I in team', () => {  
  //expect('team').not.toMatch(/I/);  
  expect('team').not.toEqual(expect.stringContaining('I'));  
});  
  
test('but there is a "stop" in Christoph', () => {  
  //expect('Christoph').toMatch(/stop/);  
  expect('Christoph').toEqual(expect.stringContaining('stop'));  
});
```

- **Asymmetric matchers**

- Estructuras de datos: arrayContaining(...)

4.test.js

```
test('Has at least expected elements', () => {  
  expect(getArray())  
    .toEqual(expect.arrayContaining(['Alice', 'Bob']));  
});  
  
test('Does not have all elements', () => {  
  expect(getOtherArray())  
    .not.toEqual(expect.arrayContaining(['Alice', 'Bob']));  
});  
  
test('Does not have all elements', () => {  
  expect(getOtherArray())  
    .toEqual(expect.not.arrayContaining(['Alice', 'Bob']));  
});
```

- **Asymmetric matchers**
 - Objetos: `objectContaining(...)`

5.test.js

```
test('Object contains at least props with values', () => {  
  expect(someObject()).toEqual(  
    expect.objectContaining({x: 3, y: 0})  
  );  
});
```

- **Asymmetric matchers**

- Los matchers se pueden combinar para crear comprobaciones complejas

6.test.js

```
test('Object contains at least props of types', () => {  
  expect(someObject()).toEqual(  
    expect.objectContaining({  
      x: expect.any(Number),  
      y: expect.any(Number),  
      values: expect.arrayContaining([expect.any(Number)])  
    })  
  );  
});
```

- **Matchers definidos por el programador**
 - Pueden ser de ambos tipos:
 - `expect(...).toXX()`
 - Asymmetric matchers
 - Existen librerías que amplían los matchers disponibles en Jest

<https://github.com/jest-community/jest-extended>

- Introducción
- **Pruebas unitarias**
 - Introducción
 - Casos de Test
 - Aserciones
 - **Tests asíncronos**
 - Dobles
 - Conclusiones

- **Async/await**

- Si se utiliza **async/await**, los tests de código asíncrono no tienen nada de especial
- Se usa **await** en el test y la *arrow function* se define **async**

1.test.js

```
test('Async Name is pepe', async () => {  
    expect(await getNameFromServer()).toBe('pepe');  
});
```

• Callback

- Para testear una función con **callback**, es necesario indicar a Jest que el test no acaba hasta que el callback haya sido llamado

2.test.js

```
test('WRONG!! Callback Name is pepe', () => {  
  getNameFromServer(name => {  
    expect(name).toBe('pepe');  
  });  
});
```

MAL!!! NO HACER

• Callback

- Para indicar que el test ha terminado se usa una función **done** que se recibe como parámetro en el test

2.test.js

```
test('Callback Name is pepe', done => {  
  getNameFromServer(name => {  
    expect(name).toBe('pepe');  
    done();  
  });  
});
```

- Promesas

- Se puede usar **done()** en el **then()**

3.test.js

```
test('Async Name is pepe', done => {  
  getNameFromServer().then(name => {  
    expect(name).toBe('pepe');  
    done();  
  });  
});
```

- **Promesas**

- Se puede usar un **matcher** para promesas (resolve)

3.test.js

```
test('Async Name is pepe (resolves)', () => {  
    return expect(getNameFromServer()).resolves.toBe('pepe');  
});
```

• Promesas

- Se puede devolver la promesa y Jest esperará a que se resuelva

3.test.js

```
test('Async Name is pepe (returning promise)', () => {  
  return getNameFromServer().then(name => {  
    expect(name).toBe('pepe');  
  });  
});
```

- Introducción
- **Pruebas unitarias**
 - Introducción
 - Casos de Test
 - Aserciones
 - Tests asíncronos
 - **Dobles**
 - Conclusiones

- Los elementos de un software siempre **dependen de otros elementos** (clases, funciones, módulos...)
- Al implementar un **test** para probar una clase, a la clase se la conoce como **SUT** (***subject under test***) y a cada una de las clases de las que depende **DOC** (***depended-on component***)
- Si queremos testear la clase en su contexto real, podemos tener **bastantes limitaciones**

- **Limitaciones de probar un SUT con sus dependencias reales (I):**
 - **Lentas:** Realiza cálculos extensivos, acceso a bases de datos o ficheros.
 - **No deterministas:** Suministra al SUT valores de entrada no deterministas, descontrolados (aleatorio, hora, señal de sensor, ...).
 - **Difícil de automatizar:** Porque es una interfaz de usuario o porque se quieren simular comportamientos difíciles de conseguir (poco ancho de banda, fallo 500 es un servidor...)

- **Limitaciones de probar un SUT con sus dependencias reales (II):**
 - **Preparación costosa:** Porque su construcción es costosa en tiempo, memoria, CPU.
 - **Acopladas a otros sistemas:** Es parte de otro componente (p.e. librería de acceso a otro servidor vía tcp/ip) o no debe cambiar su estado (p.e. un servicio de correo)
 - **No están disponibles:** porque su implementación no está disponible

- Para solventar estos problemas se usan sustitutos llamados **dobles**
- Los **dobles** son objetos que se usan como **sustitutos de las dependencias reales** cuando estos no son convenientes para las pruebas automáticas
- El nombre de **doble** (*double*) proviene de los **dobles de las películas**, que se usan en las escenas en las que no es conveniente que participe el actor real

Dobles



- Los dobles pueden **adaptarse** a las necesidades del test
 - Ejecución más rápida y sin dependencias externas
 - Comportamiento concreto (para probar cierto comportamiento en el SUT)
 - Verificación que se ha llamado a sus métodos con ciertos parámetros

- **Tipos de dobles (I):**
 - **Dummy:** Dependencia cuyo funcionamiento no afectará a las comprobaciones del test, pero se utiliza para que el código pueda ejecutarse
 - **Fake:** Dependencia que se comporta como la real, pero con una implementación más simple y más rápida para tests (p.e. Base de datos en memoria)
 - **Spy:** Proxy de una dependencia real que permite verificar en el test qué métodos han sido llamados, con qué parámetros y valor devuelto

<https://martinfowler.com/articles/mocksArentStubs.html>

- **Tipos de dobles (II):**
 - **Stubs:** Dependencia cuyo comportamiento se define en el test con respuestas predefinidas para sus métodos.
 - **Mocks:** Iguales a los stubs, pero que además se verifica que las funciones o métodos especificados han sido llamados

Aunque sea incorrecto, en muchos contextos a los **stubs** también se les llama **mocks** (aunque no se verifique que los métodos/funciones haya sido llamados)

<https://martinfowler.com/articles/mocksArentStubs.html>

- ¿Qué tipo de dependencia se puede sustituir por un doble?
 - **Una función:** Para definir su valor devuelto cuando sea invocada en el test
 - **Un objeto:** Para definir el comportamiento de sus métodos en el test.
 - **Una clase:** Para definir el comportamiento de sus métodos estáticos y los objetos creados con new
 - **Elementos exportados de un módulo:** Funciones, objetos y clases.

- ¿Cómo se usa un objeto mock en un test?
 - 1) Se define una **variable** para el objeto mock
 - 2) Se crea el objeto mock
 - 3) Se **define el comportamiento** de sus métodos
 - 4) Se **pasa el objeto al SUT** en el constructor o en un método
 - 5) Se **ejercita** el SUT (para que use el objeto mock)
 - 6) Se verifica que el **SUT se comporta como se espera**
 - 1) Se verifica **que los métodos del objeto han sido llamados con los parámetros esperados**

- **Ejemplo test GestorNotas**

- Queremos testear la clase **GestorNotas** que permite obtener la **nota media** de los alumnos
- Obtiene los alumnos de una **BaseDatosAlumnos** configurada en el constructor
- Usa el método **baseDatos.getNotasAlumno(id)** para obtener las notas de un alumno (en forma de array) para calcular su nota media
- **Sustituimos la base de datos de alumnos por un stub**

• Ejemplo test GestorNotas

gestorNotas.js

```
module.exports = class GestorNotas {  
  
  constructor(alumnos) {  
    this.alumnos = alumnos;  
  }  
  
  calculaNotaMedia(idAlumno) {  
    let notas = this.alumnos.getNotasAlumno(idAlumno);  
    let suma = 0;  
    for (let nota of notas) {  
      suma += nota;  
    }  
    return suma / notas.length;  
  }  
}
```

• Ejemplo test GestorNotas

gestorNotasStub.test.js

```
const GestorNotas = require('./gestorNotas');

test('Cálculo nota media', () => {

  const getNotasAlumno = jest.fn();

  getNotasAlumno.mockReturnValue([5, 6, 8, 9]);

  let alumnos = { getNotasAlumno };

  let gestorNotas = new GestorNotas(alumnos);

  expect(gestorNotas.calculaNotaMedia(1)).toBeCloseTo(7);


});
```

• Ejemplo test GestorNotas

gestorNotasStub.test.js

```
const GestorNotas = require('./gestorNotas');  
test('Cálculo nota media', ()=>{  
  const getNotasAlumno = jest.fn();  
  getNotasAlumno.mockReturnValue([5, 6, 8, 9]);  
  let alumnos = { getNotasAlumno };  
  let gestorNotas = new GestorNotas(alumnos);  
  expect(gestorNotas.calculaNotaMedia(1)).toBeCloseTo(7);  
});
```

Creamos un stub
de una función con
jest.fn()




• Ejemplo test GestorNotas

gestorNotasStub.test.js

```
const GestorNotas = require('./gestorNotas');  
test('Cálculo nota media', ()=>{  
  const getNotasAlumno = jest.fn();  
  getNotasAlumno.mockReturnValue([5, 6, 8, 9]);  
  let alumnos = { getNotasAlumno };  
  let gestorNotas = new GestorNotas(alumnos);  
  expect(gestorNotas.calculaNotaMedia(1)).toBeCloseTo(7);  
});
```

Configuramos el
valor que devuelve
esa función




• Ejemplo test GestorNotas

gestorNotasStub.test.js

```
const GestorNotas = require('./gestorNotas');  
  
test('Cálculo nota media', () => {  
  const getNotasAlumno = jest.fn();  
  getNotasAlumno.mockReturnValue([5, 6, 8, 9]);  
  let alumnos = { getNotasAlumno };  
  let gestorNotas = new GestorNotas(alumnos);  
  expect(gestorNotas.calculaNotaMedia(1)).toBeCloseTo(7);  
});
```

Creamos un objeto
con esa función
como método




• Ejemplo test GestorNotas

gestorNotasStub.test.js

```
const GestorNotas = require('./gestorNotas');  
  
test('Cálculo nota media', () => {  
  const getNotasAlumno = jest.fn();  
  
  getNotasAlumno.mockReturnValue([5, 6, 8, 9]);  
  
  let alumnos = { getNotasAlumno };  
  
  let gestorNotas = new GestorNotas(alumnos);  
  
  expect(gestorNotas.calculaNotaMedia(1)).toBeCloseTo(7);  
});
```

Pasamos el objeto
al SUT



- Se pueden definir diferentes **valores de retorno** en las funciones mock

```
const myMock = jest.fn();  
console.log(myMock());  
// > undefined
```

```
myMock  
  .mockReturnValueOnce(10)  
  .mockReturnValueOnce('x')  
  .mockReturnValue(true);
```

```
console.log(myMock(), myMock(), myMock(), myMock());  
// > 10, 'x', true, true
```

- Se pueden definir **el comportamiento** de las funciones mock

```
const myMock = jest.fn(() => 'default')  
  .mockImplementationOnce(() => 'first call')  
  .mockImplementationOnce(() => 'second call');  
  
console.log(myMock(), myMock(), myMock(), myMock());  
// > 'first call', 'second call', 'default', 'default'
```

- Si función mock devuelve una **promesa**, podemos simplificar su implementación

```
const asyncMock = jest.fn(() => Promise.resolve(43));  
  
//Puede escribirse como  
const asyncMock = jest.fn().mockResolvedValue(43);
```

- **Verificar llamadas en mocks**

- Los mocks se diferencian de los stubs en que el propio test verifica que las funciones/métodos han sido llamados
- Se puede verificar que los parámetros usados en las llamadas son los esperados

```
// The mock function was called at least once  
expect(mockFunc).toBeCalled();
```

```
// The mock function was called at least once with the specified args  
expect(mockFunc).toBeCalledWith(arg1, arg2);
```

```
// The last call to the mock function was called with the specified args  
expect(mockFunc).lastCalledWith(arg1, arg2);
```

• Ejemplo test GestorNotas

gestorNotasMock.test.js

```
const GestorNotas = require('./gestorNotas');

test('Cálculo nota media', () => {

  const getNotasAlumno = jest.fn();

  getNotasAlumno.mockReturnValue([5, 6, 8, 9]);

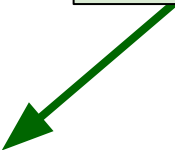
  let alumnos = { getNotasAlumno };

  let gestorNotas = new GestorNotas(alumnos);

  expect(gestorNotas.calculaNotaMedia(1)).toBeCloseTo(7);
  expect(alumnos.getNotasAlumno).toBeCalledWith(1);

});
```

Matcher para
verificar que la
función ha sido
llamada con
parámetros
concretos



- **Verificar llamadas en mocks**
 - Los matchers se pueden usar para analizar las características de los parámetros de los mocks

```
// Parámetro no es undefined ni null
expect(mock).toBeCalledWith(expect.anything());

// Parámetro es un objeto con propiedades x,y numéricas
expect(onPress).toBeCalledWith(
  expect.objectContaining({
    x: expect.any(Number),
    y: expect.any(Number),
  }),
);
```

- **Información sobre la llamada en .mock**
 - Se puede obtener **más información** de las llamadas realizadas a los mocks

```
// The mock function is called twice
expect(myMock.mock.calls.length).toBe(2);

// The first argument of the first call to the function was 0
expect(myMock.mock.calls[0][0]).toBe(0);

// The first argument of the second call to the function was 1
expect(myMock.mock.calls[1][0]).toBe(1);

// The return value of the first call to the function was 42
expect(myMock.mock.results[0].value).toBe(42);
```


• Ejercicio 5

- Se está implementando una **aplicación web de chat** con websockets
- Se tiene implementada la **clase chat** que contiene los usuarios del chat
- Cada vez que un usuario envía un **mensaje** al chat, este se reenvía a todos los demás usuarios
- Cada vez que un usuario se añade o deja el chat, se envía una **notificación** al resto de usuarios

• Ejercicio 5

- Los usuarios se **añaden y eliminan** del Chat con:
 - addUser(user)
 - removeUser(user)
- Las **notificaciones** se envían a un usuario invocando los siguientes métodos en el objeto usuario:
 - onMessage(chatName, userName, msg)
 - newUserInChat(chatName, userName)
 - userExitedFromChat(chatName, userName)

```
module.exports = class Chat {  
  
  constructor(name) {  
    this.name = name;  
    this.users = [];  
  }  
  
  addUser(user) {  
    for (let u of this.users) {  
      u.newUserInChat(this.name, user.name);  
    }  
    this.users.push(user);  
  }  
  
  removeUser(user) {  
    this.users.splice(this.users.indexOf(user), 1);  
    for (let u of this.users) {  
      u.userExitedFromChat(this.name, user.name);  
    }  
  }  
  
  sendMessage(user, message) {  
    for (let u of this.users) {  
      if (u !== user) {  
        u.onMessage(this.name, user.name, message);  
      }  
    }  
  }  
}
```

• Ejercicio 5

- Se pide implementar un **test unitario** de la clase **Chat**
- Hay que implementar **usuarios mock** y **verificar** que sus métodos son invocados con los parámetros adecuados cuando se realizan las acciones en la clase Chat:
 - Añadir un usuario
 - Eliminar un usuario
 - Enviar un mensaje

• Ejercicio 6

- Se quiere ampliar el chat para que se pueda establecer videoconferencias con un MediaServer
- El MediaServer tiene los siguientes métodos:
 - `boolean allowMoreUsers()`
 - `addUser()`
 - `removeUser()`

- ## Ejercicio 6

- Cuando se vaya a añadir un usuario a un chat, se deberá preguntar al **MediaServer si tiene capacidad.**
- En caso de que no tenga capacidad, el método Chat.addUser deberá devolver una **excepción**

• Ejercicio 6

- Se pide ampliar la clase chat para que use un MediaServer
- Hay que implementar los tests correspondientes que verifiquen que el MediaServer es usado como debería y que su comportamiento afecta al comportamiento del chat

- **Mocks de módulos**

- Cuando el **SUT es una clase** y su dependencia se recibe en el constructor, se pueden mockear un objeto o función y pasarla como parámetro
- Cuando el **SUT es una función o método** y su dependencia se recibe como parámetro, se pueden mockear un objeto o función y pasarla como parámetro
- ¿Y si el SUT importa la **dependencia como un módulo**?
- ¿Cómo se mockea un módulo?

- Ejemplo test Petición REST

users.js

```
const axios = require('axios');

module.exports = class Users {

  static async all() {
    let resp = await axios.get('/users.json');
    return resp.data;
  }
}
```

• Ejemplo test Petición REST

users.test.js

```
const axios = require('axios');
const Users = require('./users');

jest.mock('axios');

test('should fetch users', async () => {

  const users = [{ name: 'Bob' }];
  const resp = { data: users };

  axios.get.mockResolvedValue(resp);

  expect(await Users.all()).toEqual(users);
});
```

Al ejecutar esta sentencia,
el módulo axios se
convierte en un mock

Sus elementos se
mockean como los
creados con jest.fn()

- **Mocks de clases en módulos**
 - ¿Qué ocurre si el SUT tiene como dependencia un **objeto que instancia internamente** (no se le pasa como parámetro)?
 - Si la **clase** de ese objeto se **importa** de otro módulo, Jest puede **mockear el módulo** para que el objeto creado por la clase sea un **mock**

<https://jestjs.io/docs/en/es6-class-mocks>

• Ejemplo con dependencia interna

gestorNotas.js

```
const BDAlumnos = require('./bDAlumnos');

module.exports = class GestorNotas {

  constructor() {
    this.alumnos = new BDAlumnos();
  }

  calculaNotaMedia(idAlumno) {
    let notas = this.alumnos.getNotasAlumno(idAlumno);
    let suma = 0;
    for (let nota of notas) {
      suma += nota;
    }
    return suma / notas.length;
  }
}
```

• Ejemplo con dependencia interna

gestorNotas.test.js

```
const DBAlumnos = require('./bDAAlumnos');
const GestorNotas = require('./gestorNotas');

jest.mock('./bDAAlumnos');

test('Cálculo nota media', () => {

  const getNotasAlumno = jest.fn().mockReturnValue([5, 6, 8, 9]);

  let mockAlumnos = { getNotasAlumno }

  DBAlumnos.mockImplementation(() => mockAlumnos);

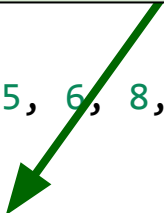
  let gestorNotas = new GestorNotas();

  expect(gestorNotas.calculaNotaMedia(1)).toBeCloseTo(7);

  expect(mockAlumnos.getNotasAlumno).toBeCalledWith(1);

});
```

Redefinimos la implementación del constructor de DBAlumnos para que devuelva un mock



gestorNotas2.test.js

```
const DBAlumnos = require('./bDAlumnos');
const GestorNotas = require('./gestorNotas');

jest.mock('./bDAlumnos');

test('Cálculo nota media', () => {
  DBAlumnos.mockImplementation(() =>
    ({ getNotasAlumno: jest.fn().mockReturnValue([5, 6, 8, 9]) })
  );

  let gestorNotas = new GestorNotas();
  let gestorNotas2 = new GestorNotas();

  expect(gestorNotas.calculaNotaMedia(1)).toBeCloseTo(7);

  expect(DBAlumnos.mock.results[0].value.getNotasAlumno).toBeCalledWith(1);
  expect(DBAlumnos.mock.results[1].value.getNotasAlumno).not.toBeCalled();
});
```

Si queremos que devuelva mocks diferentes cada vez que se llama a new no se pueden usar variables locales al test

Podemos acceder a los objetos creados en cada llamada con .mock

ejem8

- Introducción
- **Pruebas unitarias**
 - Introducción
 - Casos de Test
 - Aserciones
 - Tests asíncronos
 - Dobles
 - **Conclusiones**

- Existen **frameworks** que facilitan la implementación de pruebas automáticas
- Son **muy similares** independientemente del **lenguaje de programación** usado
- Cada test tiene 3 partes diferenciadas
 - **Given – When – Then**
 - **Arrange – Act – Assert**

- Las **aserciones** son una parte importante de las librerías porque hacen el **test más legible**
- Los **dobles** de las dependencias **facilitan** la creación de pruebas automáticas del SUT

- La librería **Jest** ofrece más funcionalidades de las que hemos visto
 - **Snapshot testing:** Comparar el resultado de una función con un valor guardado de una ejecución previa
 - **Manipular el tiempo:** Permite cambiar en los tests el comportamiento de `setTimeout`, `setInterval`, etc.
 - **Ejecución de tests en paralelo:** Para aprovechar la potencia de la máquina
 - **Control de cobertura del código testeado**

- **Mocks de módulos:**

- Permite implementar mocks de módulos que pueden ser usados en diferentes tests (automatic mocks)
- Permite mockear sólo algunos elementos de un módulo y usar las implementaciones reales de otros elementos
- Permite mockear librerías internas de Node.js