

Test Angular

# CONTENIDOS

1. Teoría del Testing
2. Testing en Angular

# Verificación de Aplicaciones

- Garantizar que la aplicación, dentro de las posibilidades reales, se encuentra libre de errores.
- Para ello se realiza un conjunto finito de **pruebas**.
- **Las pruebas** son una serie de actividades pre-diseñadas para permitir encontrar, localizar y modificar los posibles errores que tenga el diseño, aparte de probar el correcto funcionamiento de la aplicación.

Curso de QA: <https://youtu.be/LWw4nLNVrsw>

# Objetivos de los procesos de pruebas

- Detectar posibles defectos en el *software*.
- Verificar la integridad de los componentes que forman la aplicación o proyecto.
- Verificar la correcta implementación de los requisitos.
- Identificar posibles errores de *software* antes de entregar el proyecto al cliente.
- Diseñar procesos de prueba que sirvan para obtener posibles errores con el menor tiempo y esfuerzo (lo cual requerirá una amplia experiencia como programador de procesos de prueba).

# Principios de los procesos de pruebas

1. Sirven para demostrar que existen defectos en el *software*
  - *Si las pruebas no detectan errores no significan que no existan*
2. No es posible realizar una prueba que cubra todas las variables
3. Se realizan al inicio del ciclo de vida de los productos
  - *Planificación-Análisis-Diseño-Implementación-Instalación-Uso/Mantenimiento*
4. Agrupa las pruebas por tipo para revisar el *software*

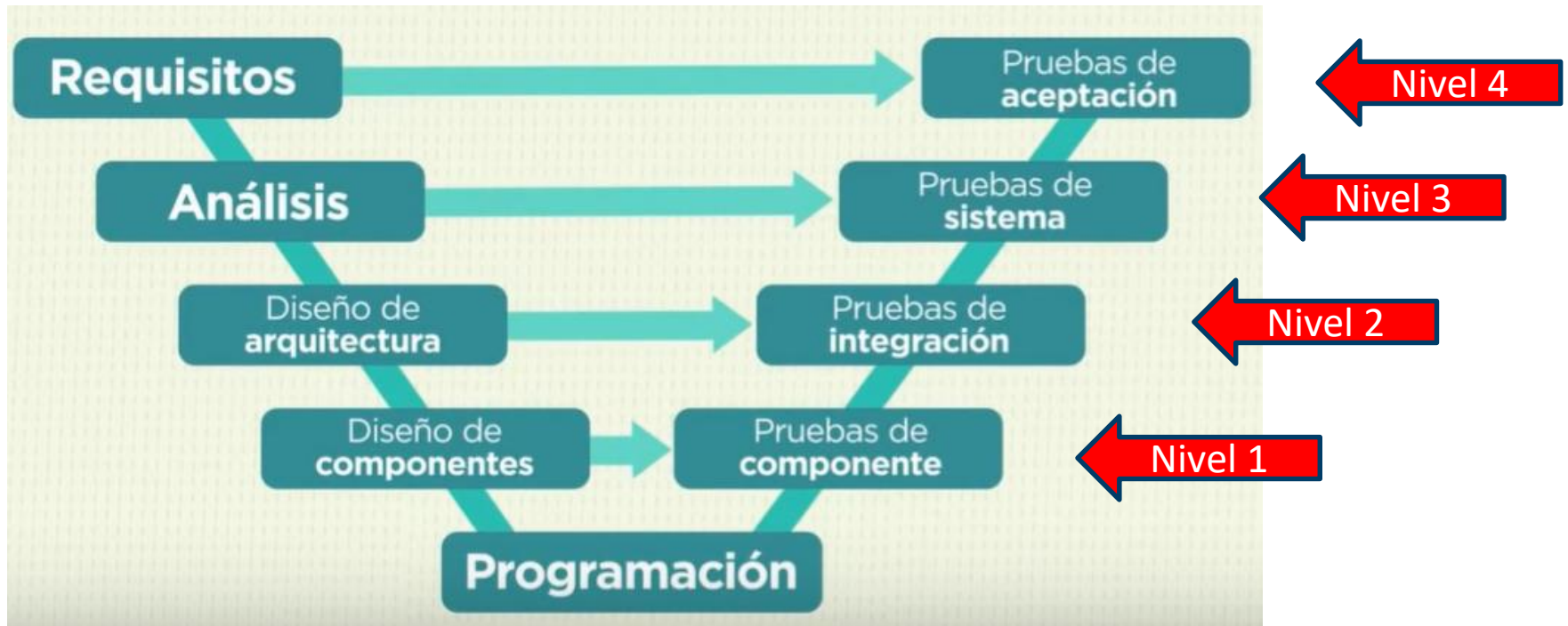
<i>Módulo 1:</i>	<i>Módulo 2:</i>	<i>Módulo 3:</i>
<i>Unitarias</i>	<i>De Sistema</i>	<i>De GUI</i>
<i>Integración</i>	<i>De Desempeño</i>	<i>De Configuración</i>
<i>Regresión</i>	<i>De Carga</i>	<i>De Compatibilidad y Conversión</i>
<i>De Humo</i>	<i>De Volumen</i>	<i>De Múltiples Sitios</i>
	<i>De Recuperación</i>	<i>De Seguridad de Acceso</i>
		<i>De Estilo</i>
5. Deben actualizarse periódicamente
  - *Para detectar nuevos errores*
6. Se realizan dependiendo del funcionamiento del *software*
  - *No es lo mismo la seguridad de una aplicación bancaria que de una aplicación de compra de entradas.*
7. El producto final debe cumplir con las expectativas del usuario

# Modelos de pruebas

- Forma de aplicar los distintos modelos de pruebas para software
- Se verifican los objetivos y se utiliza el modelo que más convenga:
  - Modelo en V, W,..,

# Método de pruebas en V

- Cuatro niveles de prueba:



# Modelo en V: Proceso de desarrollo iterativo-incremental

- Se forma un grupo de tareas dentro del sistema y sirve para probarlo
- Las Iteraciones son el número de veces que se realiza una prueba modificando algunas condiciones. Cada tarea debe realizarse las veces que sean necesarias.
- Incremental porque no se puede pasar otra prueba hasta que no se haya cumplido la anterior.



# Niveles de pruebas: Pruebas de componentes

- Objetivo: probar el funcionamiento de componentes de software de forma neutral enfocadas a los requisitos de los componentes.
- Ejemplo: Software de una compañía telefónica con los siguientes módulos:
  - Inicio de sesión
  - Saldo pendiente
  - Pagos y recargas
  - Historial de llamadas
- En inicio de sesión se prueba como se comporta el sistema cuando se introducen datos correctos, incorrectos o se dejan campos vacíos

# Niveles de pruebas: Pruebas de integración

- Objetivo: En base a la arquitectura del sistema o a tareas funcionales para facilitar las tareas de integración y disminuir los riesgos
- Se comprueba el flujo de información entre módulos. En el ejemplo de la compañía telefónica, relación entre el módulo de saldo pendiente y el de pagos y recargas. Si el cliente tiene una deuda de 10€ y se dirige al módulo de pagos y recarga 10€ el saldo de llamadas debería seguir en 0€.

**2 = PRUEBAS UNITARIAS**

**0 = PRUEBAS DE INTEGRACIÓN**



# Niveles de pruebas: Pruebas de sistemas

- También pruebas de validación. Revisan el funcionamiento del software en su totalidad con el objetivo de que cumpla con los requisitos funcionales y no funcionales y así minimizar la posibilidad de errores
- Aquí aparecerían las pruebas funcionales, pruebas de carga, de estrés, etc
- Requieren la documentación completa de la aplicación (manuales de usuario, manuales de explotación, manuales de especificaciones técnicas, etc.) y configuración del sistema acorde a las necesidades de la aplicación. Dentro de este tipo de pruebas, destacan dos fundamentalmente:
  - **Pruebas alfa:** se realizan en el entorno donde se desarrolla la aplicación, en presencia del cliente, y es un conjunto de pruebas que antes de ser aplicadas a la aplicación ya conocemos el resultado que tienen que arrojar. Este tipo de pruebas suelen prolongarse en el tiempo (varios días si se habla de aplicaciones potentes).
  - **Pruebas beta:** se realizan en el entorno donde el usuario está explotando o trabajando con la aplicación. Si se pone de ejemplo a Microsoft se puede observar como lanza al mercado versiones beta de ciertos productos con el objetivo de que sus clientes consumidores las prueben en sus equipos e informen (bien manual o automáticamente) sobre los errores que van produciendo. Suelen ser de mayor duración que las alfa.

# Niveles de pruebas: Pruebas de aceptación

- El cliente determina si el *software* tiene éxito en función de su uso y su comportamiento

# Clasificación de pruebas

- Más de 100 pruebas diferentes. Las más importantes son:
  - Pruebas **funcionales**:
    - pueden aplicarse en cualquier nivel del proceso.
    - Verifican que el software opera según las especificaciones
    - Dos modalidades, dependiendo de la forma en que se ejecuten:
      - **Manuales**: requiere de la intervención en la prueba (como si se fuera un usuario que está probando la aplicación) para garantizar que hace lo que debe. Se llevará a cabo el proceso diseñado con la intervención y, si se producen errores, se detallarán una vez se haya finalizado dicho proceso
      - **Automáticas**: son justamente lo contrario a las anteriores, no requieren de la intervención y lo que hay con ello es un ahorro de tiempo considerable

# Clasificación de pruebas

- Pruebas no **funcionales**:
  - Se aplican en cualquier nivel del proceso después de las pruebas funcionales.
  - Se comprueban parámetros como:
    - Desempeño
    - Escalabilidad
    - Portabilidad
    - Estabilidad
    - Seguridad
    - Operatividad

# Clasificación de pruebas

- Pruebas de **caja blanca o estructurales**:
  - Mejoran seguridad, usabilidad y robustez del sistema
  - Basadas en el funcionamiento interno del *software*
    - Se verifican fallos en la seguridad
    - Trayectorias mal estructuradas o rotas en los procesos de codificación
    - Flujo de valores de entrada a través del código y los resultados esperados
    - Funcionalidad de los bucles condicionales

# Clasificación de pruebas

- Pruebas de **caja negra ,de detección de errores o e2e**:
  - Verifican la funcionalidad del software sin examinar la estructura del código interno.
  - Sólo se tienen en cuenta parámetros de entrada y de salida.
  - Se realizan los siguientes pasos:
    - Se reconocen los requerimientos y especificaciones que hay que probar.
    - Se seleccionan valores válidos e inválidos para ver como son procesados por el software
    - Se determina cuáles son las respuestas esperadas para cada valor elegido
    - Se construyen casos de prueba para cada valor y se ejecutan



# Clasificación de pruebas

- Pruebas de **regresión o de choque**:
  - Se ejecutan para confirmar que los cambios hechos en el código no han afectado a otras funciones

# Pruebas de aplicaciones Web

- Pruebas de Enlaces: Todos los enlaces de la página funcionan
- Pruebas de formatos: Forma en la que el sitio Web consigue información de los usuarios para mantener la interacción. Se deben probar las respuestas del sistema a valores incorrectos o nulos
- Pruebas de cookies
- Pruebas de bbdd: verificar integridad de los datos
- Pruebas de navegación.
- Pruebas de contenido: lógico, entendible y errores ortográficos
- Pruebas de interfaces, envían mensajes apropiados
- Pruebas de navegador
- Pruebas de sistema operativo
- Pruebas de Carga: Múltiples conexiones a bbdd
- Pruebas de estrés
- Pruebas de seguridad: Reacción a valores inválidos. Pruebas captcha. Protocolo ssl.

# Casos de Prueba: GOOGLE y sus pruebas de 10'

- “Cada 10 minutos cambiaba de aplicación. Y a medida que los minutos pasaban mi equipo iba entendiendo el mensaje: descartaron la idea de describir el problema, la aplicación o cualquier cosa innecesaria. Cambiaron prosa por listas de viñetas. Caso omiso a cualquier información no relevante para el testing. Si no era importante, lo ignoraban. No había tiempo.”
- Así creamos el método ACC (atributos, componentes y capacidades)
  - **Atributos** (valor del producto para el usuario)
  - **Componentes** (enumerar, de memoria, los componentes del software)
  - **Capacidades** (las acciones que realiza el sistema)
- Caso de estudio: Cómo hace Google sus planes de pruebas (y los hace en menos de 10 minutos) [http:// www.javiergarzas.com/2012/12/google-pruebas.html](http://www.javiergarzas.com/2012/12/google-pruebas.html)

# Casos de Prueba: POKER ONLINE APP

- Real Poker Online es una app cliente servidor desarrollada en Java que permite a jugadores de todo el mundo jugar poker
- Se busca validar los requerimientos, además de la escalabilidad del proyecto (Eclipse IDE, Ant, JUnit, XDoclet, Hibernate JDO, Tapestry Web Application Framework, Hessian RPC, Sun ONE J2EE 1.3 certified application server, Firebird RDBMS y Tomcat)
- La estimación inicial para las pruebas hecha por el cliente era de 8000 horas-hombre
- Bughuntress propuso pruebas automatizadas en vez de un proceso manual: caja negra, simulación, funcionalidad, estrés, usabilidad y pruebas de apuestas lógicas. Las pruebas finalmente requirieron 2500 horas-hombre
- CASE STUDIES: ONLINE GAME: <http://www.bughuntress.com/portfolio/case-studies/poker-online>

# Herramienta de Automatización

## Selenium

Selenium es uno de los frameworks más utilizados para probar aplicaciones web, principalmente para la interfaz web y las pruebas funcionales. Viene con una serie de herramientas como Selenium IDE, Selenium RC, Selenium WebDriver y Selenium Grid que ofrece diferentes soluciones para atender diferentes requisitos de automatización de pruebas.

## Referencias

<http://www.seleniumhq.org/projects/webdriver/>

# Herramienta de Automatización

## Appium

Appium es un framework de automatización de pruebas para probar aplicaciones web nativas, híbridas y móviles para plataformas iOS, Android y Windows en dispositivos reales y simuladores. Dado que soporta aplicaciones multiplataforma, permite probar aplicaciones en diferentes plataformas utilizando la misma API. Appium permite a los usuarios elegir el idioma que tiene las bibliotecas de clientes de Selenium como Java, Objective-C, JavaScript con Node.js, PHP, Ruby, Python, C # etc. para crear pruebas.

## Referencias

<http://appium.io>

# Herramienta de Automatización

## JMeter

JMeter es una herramienta basada en Java diseñada para cargar el comportamiento de la aplicación y medir el rendimiento del sitio web. Puede probar recursos estáticos y dinámicos que incluyen servicios web SOAP / REST, sitios web HTTP y HTTPS, bases de datos, FTP y servidores de correo, así como PHP, ASP.NET y Java. Funciona simulando la carga en el servidor para analizar el rendimiento general de la aplicación / sitio web bajo prueba.

## Referencias

<http://jmeter.apache.org>

# Herramienta de Automatización

## Jenkins

Jenkins es una herramienta para iniciar pruebas continuas y construir la integración a través de la automatización. Proporciona una forma poderosa de administrar los cambios de código, las pruebas y el ciclo de vida del despliegue, junto con la administración de releases, acelerando el ciclo de vida general del desarrollo del software. Hoy en día, Jenkins ofrece soporte para más de 1.200 plugins que le permiten integrarse con cualquier tecnología.

## Referencias

<https://jenkins.io>



# Herramienta de Automatización

## Testlink

TestLink es una herramienta de gestión de pruebas basada en la web ampliamente utilizada. Proporciona soporte para administrar y mantener casos de prueba, conjuntos de pruebas, documentos de prueba y proyectos en un solo lugar. Puede alojarse en un servidor e integrado con herramientas de seguimiento de errores como Mantis, JIRA, Bugzilla, FogBugz, etc. para facilitar el proceso de ejecución de pruebas. TestLink se puede utilizar tanto para pruebas manuales como automatizadas.

## Referencias

<http://testlink.org>

# Herramienta de Automatización

## PhantomJS

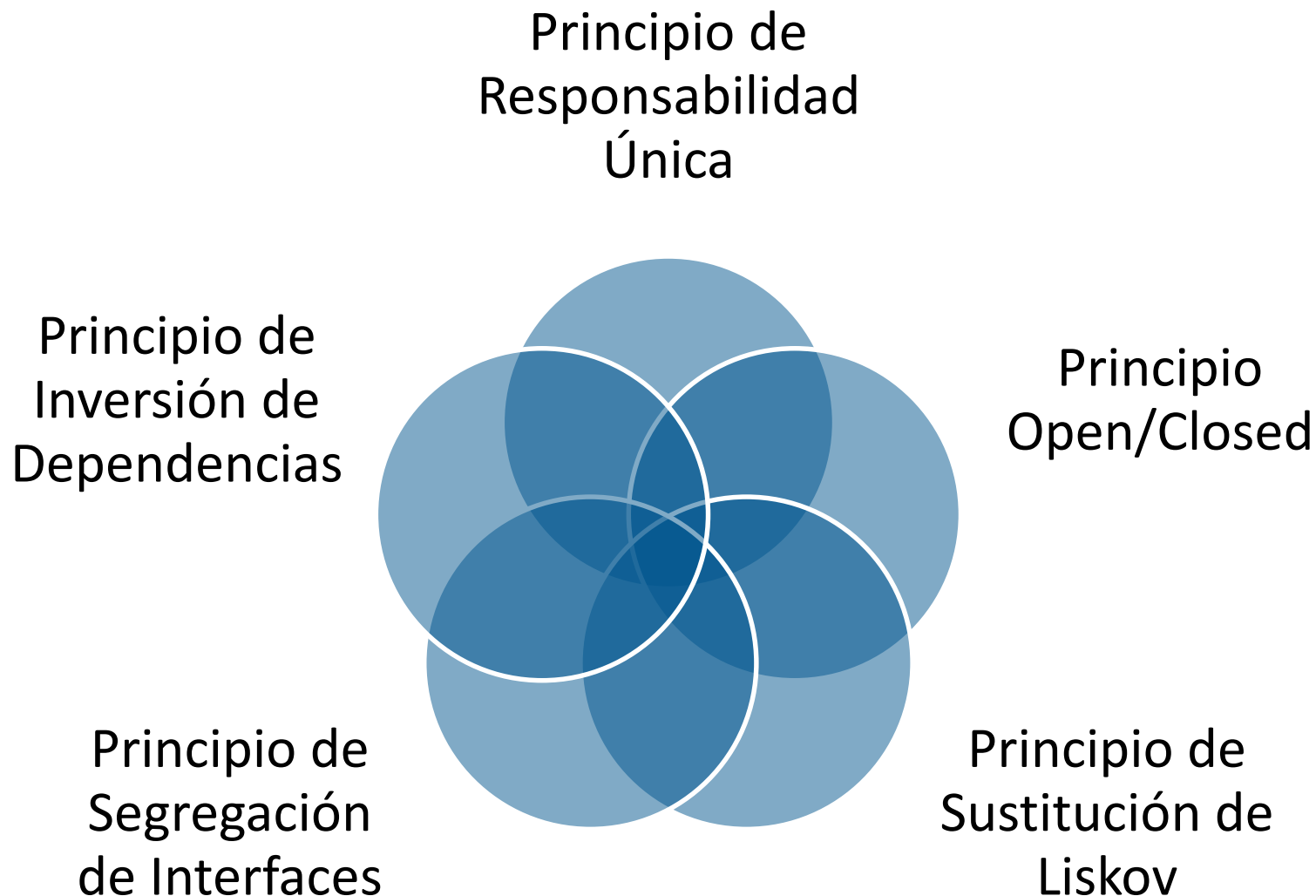
PhantomJS es un navegador que se utiliza para automatizar las interacciones de la página con fines de prueba. Ayuda a los usuarios a habilitar la navegación y el comportamiento del usuario en una página sin cargar la interfaz gráfica.

PhantomJS imita y manipula una página web para llevar a cabo la automatización de pruebas que en última instancia, ahorra una tremenda cantidad de tiempo para los probadores.

## Referencias

<http://phantomjs.org>

# Principios SOLID



# Principios SOLID: Responsabilidad Única (SRP)

- **Un objeto debe realizar una única cosa.** Es importante que las responsabilidades estén segregadas y las clases y métodos hagan el menor trabajo posible.
- Cuando algún punto del código hace muchas labores esto quiere decir que los resultados pueden ser varios. Esto se traduce a **testing** en muchos casos posibles de salida que hay que testar.
- Ej:

```
public class Vehicle {  
    public int getWheelCount() {  
        return 4;  
    }  
  
    public int getMaxSpeed() {  
        return 200;  
    }  
  
    @Override public String toString() {  
        return "wheelCount=" + getWheelCount() + ", maxSpeed=" + getMaxSpee  
    }  
  
    public void print() {  
        System.out.println(toString());  
    }  
}
```

**Cuando hay diferentes entradas y puede haber diferentes caminos o acciones. El número de combinaciones de estados y comportamientos al final de una ejecución crece de manera exponencial.**

# Principios SOLID: Abierto/Cerrado (OCP)

- **Una entidad debe estar abierta a extensión pero cerrada a modificación**
- Si el código crece a medida que evoluciona el software, es posible que no se estén distribuyendo bien las responsabilidades y que no se esté encapsulando lo que varía, por lo tanto habrá que modificar los tests, porque varía el código testado.

**Cambiar en numerosas ocasiones un test cuando nuestro software crece por extensión y no por modificación directa de requerimientos es un claro indicador de la violación del principio abierto/cerrado.**

# Principios SOLID: Sustitución de Liskov (LSP)

- **Si en alguna parte del código se usa una clase, y esta clase es extendida, se tiene que poder utilizar cualquiera de las clases hijas y que el programa siga siendo válido**
- Si hacemos tests sobre la clase padre que no funcionan sobre la clase hija se viola este principio y obligaría a crear tests de prueba sobre la clase hija.

**A la hora de usar la herencia, y sobre todo la sobreescritura de métodos de los supertipos ten hay que pasar test sobre lo que se va a heredar y sobreescribir y valorar si es la mejor alternativa.**

# Principios SOLID: Segregación de Interfaces (ISP)

- **Ninguna clase debería depender de métodos que no usa**
- Si existen métodos que no sirven para nada en clases, que sólo están ahí como consecuencia de implementar una interfaz de la cual sólo interesa un método, a la hora de testar esas clases o se prueba algo que no vale para nada o baja la cobertura de código testado por culpa de un mal diseño...

**Cada método de una interfaz es siempre un claro candidato a ser testado. Por lo tanto es bueno que las interfaces sean concretas y específicas. De lo contrario existirán varias implementaciones innecesarias que tendrán test inservibles.**

# Principios SOLID: Inversión de Dependencias (DIP)

- **Ninguna clase debería depender de métodos que no usa**
- El código que es el núcleo de la aplicación no debería depender de los detalles de implementación, como pueden ser el framework utilizado, la base de datos, o el modo de conexión al servidor... Todos estos aspectos se especificarán mediante interfaces, y el núcleo no tendrá que conocer cuál es la implementación real para funcionar
- En el contexto del testing sirve para establecer el alcance real de las pruebas

**Si la implementación no depende del sujeto de prueba se pueden realizar implementaciones exclusivas en tiempo de test**



# Test Unitarios o De Caja Blanca

- Se centran en **probar** que el **código**, clase por clase y método por método, **hace lo que tiene que hacer**. Comprobando que el **comportamiento** y el **estado** del sistema sean los **esperados**. Es decir, a unas entradas, tras una ejecución corresponden unas salidas o que se hayan ejecutado otras “N” cosas.
- Si el **código** está **desacoplado** del framework y de los agentes externos al máximo y la dependencia entre estos y el código es la mínima, en otras palabras, si es posible **aislar el software**; simplemente con la **cobertura** de los test unitarios se cubrirá la **mayor parte del código**.

# Test Unitarios: Conceptos

- **Sujeto bajo pruebas:** el sujeto bajo pruebas es la clase testada. Es decir, el actor principal que va a ser el objeto de todos nuestros test. En inglés se dice “Subject Under Test” y sus iniciales son SUT.
- **Cobertura de código:** es la parte de código o **caminos** de ejecución que han sido **testados**.
- **Alcance del test:** o también llamado en inglés “**Scope**” del test. Es lo que abarca en cobertura de código la ejecución de un test. Un test **unitario puro** tiene unas **fronteras** o límites que no salen más allá del propio SUT, por lo tanto su Scope es reducido.

# Test Unitarios: Tipos

### ▫ Según su **Interacción**:

- **Aislados, solitarios, no sociables:** son la base de los test unitarios, también los podemos llamar test **unitarios puros**. **El Scope de estos test es la unidad mínima testable**, es decir sólo testarán el SUT sin cruzar sus fronteras. Se empieza por ellos.
- **Sociables o en colaboración:** En estos test pueden interactuar varios de los SUT ya testados de forma que el **scope del test es bastante más amplio**. Estos test tardan más tiempo en pasar y son susceptibles de producir **fallos en cascada**, además su mantenimiento suele ser más costoso. A cambio dan una **cobertura de código más amplia**.

### ▫ Según su **Intención**:

- **De verificación de estado:** comprueban que a una entrada el resultado tras la ejecución es la salida esperada o lo que es lo mismo, un estado (variable) del sistema tras una ejecución queda en otro estado. Las comprobaciones de este tipo son las “Asserts” o “asertar”.
- **De verificación de comportamiento:** en lugar de testar en que estado ha quedado el sistema simplemente **se verifica que se han hecho una o varias llamadas, es decir, se verifica que el sistema se comporte como esperamos**. Estos test suelen identificarse con la palabra clave “Verify” o verificación.

# Test Unitarios: Estructura AAA

- **“Arrange” o preparación:** hacemos los **preparativos** para que lo que vayamos a testar esté todo lo aislado que queramos y **no cruce fronteras que no nos interesen**. Marcamos el límite de nuestro sujeto bajo pruebas y **preparamos la entrada** que le vamos a dar. En concreto **preparamos el “Escenario de test”**
- **“Act” o Llamada al método:** **ejecutamos** la acción que queremos testar en el sujeto bajo pruebas.
- **“Assert” o comprobación:** tras la ejecución **vemos que el sistema ha quedado en el estado esperado** a las entradas o que se ha comportado de la manera que esperamos porque ha disparado una o varias acciones.

# Test Unitarios: Buenas Prácticas

- Las clases de test deberían tener como nombre el sujeto bajo pruebas y como nombres de métodos el escenario en relación con lo que se espera obtener y la acción ejecutada. Ej.:

*shouldObtainEmployeeListWhenManagerIsNotEmpty()*

- Reutilizar el código de test, para hacer test mantenible y con menos esfuerzo
- Programar de modo ordenado y limpio, con frecuencia habrá que construir entidades que sean necesarias para aislar un SUT, o para emular ciertas entradas que se quieran dar.

# Test Unitarios: Uso de colaboradores

- Uso de colaboradores:
  - **Dummy:** No hace nada, esta vacío y devuelve vacío.
  - **Stub:** No hace nada más que devolver un valor por defecto.
  - **Spy:** Introducimos un parámetro al que tenemos acceso para que nos informe de algo que ocurre dentro del test.
  - **Mock:** Añadimos funcionalidad a un objeto que cumple la interfaz de SUT sólo para el propósito del test.
  - **Fake:** Implementa una funcionalidad que puede emular a la realidad porque funciona según los datos de entrada se reciban, pero no es real.
- Construcción de colaboradores:
  - **Mothers:** Es un conjunto de “Factory Methods” que nos permiten crear diferentes objetos para nuestros tests.
  - **Builders:** Es el uso del patrón de diseño “builder\*” para configurar diferentes objetos para nuestro test.

# Test Unitarios: Qué hay que testar

1. **Core de la lógica de negocio en general**, en esto se incluyen las decisiones lógicas, como los if, switch/case, que van a derivar en un camino u otro dependiendo del estado de nuestro sistema. También englobamos aquí las **operaciones matemáticas, o la algoritmia** en general que tiene lugar en nuestro sistema. Las operaciones sobre conjuntos y colecciones también son candidato fiel a ser testadas en primer lugar.
  - *prestar atención a los valores límite o no comunes.*
2. **Construcción de objetos**. Esto puede ser un punto de fallo bastante habitual, ya que es muy probable construir objetos y estructuras de datos que no son del todo consistentes o no interactúan bien, o de la forma esperada, entre los elementos que las componen.
3. El **trasiego de información entre capas**, o entre diferentes ámbitos de nuestro sistema.

# Test Unitarios: Qué no hay que testar

- Frameworks o código de terceros
- El propio lenguaje de programación
- Los getters y setters



# Test Unitarios: Por dónde empezar

- Si no vemos claro cómo empezar es porque la clase está mal diseñada de cara al testing
- Impedimentos:
  - Estáticos: **no podemos aislar una porción de código que hace uso de un estático.** El test de una funcionalidad que usa un estático está condenado a ser sociable.
  - Los métodos que **hacen más de una cosa:** la creación de colaboradores, preparación de escenarios o la cantidad de verificaciones que tenemos que hacer cuando los métodos no respetan el hacer una y solo una cosa, crecen de manera exponencial.
  - Los métodos que tienen **complejidades ciclomáticas elevadas:** también hacen que crezcan de manera exponencial los escenarios, colaboradores a preparar y la cantidad de resultados de estado y comportamiento en los cuales desembocan normalmente. **Un nivel de bucle anidado mayor a dos empieza a ser demasiado para que el SUT sea testable.**
  - Intenta evitar para no generar ruido indeseable e incomodidades de cara al testing:
    - Los **métodos privados.**
    - Los **métodos y clases finales.**
    - El uso de “new” o la **creación de dependencias** en forma de nuevos objetos.

# TDD: Test Driven Development

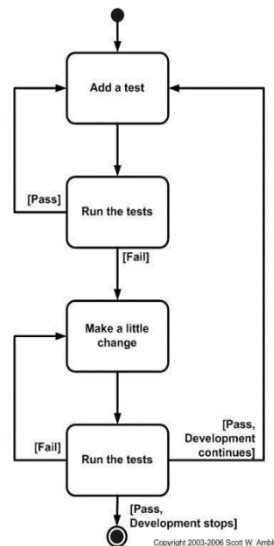
- Desarrollo guiado por pruebas. Se basa en:

**Escribir pruebas**

**Escribir código**

**Refactorización**

- Representa la alineación entre los requerimientos y las pruebas:



# Unit Testing

1. Creamos un nuevo proyecto en vsc
2. Creamos una clase nueva calculator y la importamos en AppComponent:

```
multiply(numberA: number, numberB: number): number {  
    return numberA * numberB;  
}divide(numberA: number, numberB: number): number {  
    return numberA / numberB;  
}
```

3. Creamos un método OnInit en AppComponent e introducimos el siguiente código:

```
let calculator = new Calculator(); let result = calculator.multiply(3,3); console.log(result === 9); //'Test passed'  
console.log(result !== 12); //'Test passed'  
let result2 = calculator.divide(6,2); console.log(result2 === 3); //'Test passed' console.log(result2 !== 34); //'Test  
passed'  
let result3 = calculator.divide (6,0); --prueba con flujo de datos
```

4. Nuevas pruebas y refactorizamos:

```
divide(numberA: number, numberB: number): number{  
    if(numberB === 0){  
        return null;  
    }  
    return numberA / numberB;  
}
```

- *De esta manera necesitamos el código para probar, no podemos hacer TDD*

# Unit Testing con Karma y Jasmine

- Podemos implementar TDD. Programando de esta manera se requiere tiempo de planificación previo a la programación.
- Por defecto Angular instala el servidor de pruebas Karma y realiza una serie de pruebas automáticas por componente.
- Para correr los test se teclea en el terminal: **npm test**


Lanza 3 pruebas sobre el componente

```
Chrome 69.0.3497 (Windows 10 0.0.0): Executed 0 of 3 SUCCESS (0 secs / 0 secs)
LOG: true
Chrome 69.0.3497 (Windows 10 0.0.0): Executed 0 of 3 SUCCESS (0 secs / 0 secs)
LOG: true
Chrome 69.0.3497 (Windows 10 0.0.0): Executed 0 of 3 SUCCESS (0 secs / 0 secs)
LOG: true
Chrome 69.0.3497 (Windows 10 0.0.0): Executed 0 of 3 SUCCESS (0 secs / 0 secs)
Chrome 69.0.3497 (Windows 10 0.0.0): Executed 3 of 3 SUCCESS (0.169 secs / 0.129 secs)
TOTAL: 3 SUCCESS
TOTAL: 3 SUCCESS
```



# Servidor Karma

- Archivo de configuración:

 karma.conf.js

En servidores de integración continua se incluye un emulador de interfaz gráfica

```
module.exports = function (config) {  
  config.set({  
    basePath: '',  
    frameworks: ['jasmine', '@angular-devkit/build-angular'],  
    plugins: [  
      require('karma-jasmine'),  
      require('karma-chrome-launcher'),  
      require('karma-jasmine-html-reporter'),  
      require('karma-coverage-istanbul-reporter'),  
      require('@angular-devkit/build-angular/plugins/karma')  
    ],  
    client: {  
      clearContext: false // leave Jasmine Spec Runner output visible in browser  
    },  
    coverageIstanbulReporter: {  
      dir: require('path').join(__dirname, './coverage'),  
      reports: ['html', 'lcovonly'],  
      fixWebpackSourcePaths: true  
    },  
    reporters: ['progress', 'kjhtml'],  
    port: 9876,  
    colors: true,  
    logLevel: config.LOG_INFO,  
    autoWatch: true,  
    browsers: ['Chrome'],  
    singleRun: false  
  });  
};
```

- Archivo de contexto para pruebas:

 test.ts

- karma se puede ejecutar con flags:

- `ng test --code-coverage`: genera un informe de cobertura y se almacena en una carpeta de coverage en formato html.

All files src/app

96.3% Statements 26/27 45.45% Branches 10/22 100% Functions 8/8 100% Lines 23/23

Press n or j to go to the next uncovered block, b, p or k for the previous block.

File	Statements	Branches	Functions	Lines
app.component.ts	95.24% 20/21	45.45% 10/22	100% 4/4	17/17
calculator.ts	100% 6/6	100% 0/0	100% 4/4	6/6

# Jasmine: Preparación de pruebas

### ▫ Pasos para crear un archivo de pruebas:

1. Creamos un fichero con la extensión .spec.ts
2. Añadimos el import del artefacto que queremos probar
3. Creamos una descripción para todas las pruebas y una por cada prueba.
4. Las pruebas se ejecutan dentro del descriptor “it” siguiendo la regla de las 3As (Arrange, Act, Assert)

```
import { Calculator } from './calculator';

describe('Test for Calculator', () => {

  describe('Test for multiply', () => {

    it('multiply for 3', () => {
      // Arrange
      let calculator = new Calculator();
      // Act
      let number = calculator.multiply(3, 3);
      // Assert
      expect(number).toEqual(9);
    });

    it('multiply for 4', () => {
      let calculator = new Calculator();

      let number = calculator.multiply(2, 2);

      expect(number).toEqual(4);
    });
  });
});
```

### Ejercicio 1

- Añade el test de la función `calculator.divide` que devuelve un número y vuelve a correr los test y el informe de coverage
- Añade el test de la función `calculator.divide` para los casos que devuelva un null (usa la función `toBeNull` de Jasmine).

Test+coverage

# Jasmine: API

```
it("test of matchers", ()=>{  
  let name = 'raul'  
  let name2;  
  expect(name).toBeDefined();  
  expect(name2).toBeUndefined();  
  expect(1+2 == 3).toBeTruthy();  
  expect(1+1 == 3).toBeFalsy();  
  expect(5).toBeLessThan(10);  
  expect(20).toBeGreaterThan(10);  
  expect('1234567').toMatch(/123/);  
  expect(["apples", "oranges", "pears"]).toContain("oranges")  
});
```

<https://jasmine.github.io/api/2.9/global>



### Jasmine: API: beforeEach()

- Para no repetir la instanciación del objeto, se puede invocar al principio del test a la función `beforeEach()` que haría el *arrange* de todos los test de forma común.

```
let calculator;  
  
//Arrange  
  
beforeEach(() => {  
    calculator = new Calculator();  
});
```

### Ejercicio 2

- Rehacer el código del `calculator.spec.ts` con la función `beforeEach()`

# Jasmine: API: focus test

- Se puede centrar la ejecución de determinadas pruebas incorporando una “f” delante del test que se quiere correr. Puede ser a nivel de clase, de *describe* o de *it*:

```
fdescribe('Test for Calculator', ()  
=> {...});  
  
fit("multiply for 3", ()=>{...});
```

### Ejercicio 3(I)

- Requerimientos:
  - Diseñar una clase persona, que tenga los siguientes atributos:
    - nombre, apellido, talla, peso, altura, edad
  - Debe calcular lo siguiente:
    - calcular IMC: (Indice de masa corporal)

$$\text{IMC} = \frac{\text{Peso (kg)}}{\text{Estatura}^2 (\text{Mts.})}$$



Índice de Masa Corporal (IMC)	Clasificación
Menor a 18	Peso bajo. Necesario valorar signos de desnutrición
18 a 24.9	Normal
25 a 26.9	<b>Sobrepeso</b>
Mayor a 27	<b>Obesidad</b>
27 a 29.9	<b>Obesidad grado I.</b> Riesgo relativo <b>alto</b> para desarrollar enfermedades cardiovasculares
30 a 39.9	<b>Obesidad grado II.</b> Riesgo relativo <b>muy alto</b> para el desarrollo de enfermedades cardiovasculares
Mayor a 40	<b>Obesidad grado III Extrema o Mórbida.</b> Riesgo relativo <b>extremadamente alto</b> para el desarrollo de enfermedades cardiovasculares

### Ejercicio 3 (II)

- Diseñar los siguientes casos de prueba:

Peso (kg)	Altura (m)	IMC	Result
40	1.65	14	'down'
58	1.65	21	'normal'
68	1.65	25	'overweight'
75	1.65	27	'overweight level 1'
90	1.65	33	'overweight level 2'
120	1.65	44	'overweight level 2'

Peso (kg)	Altura (m)	IMC	Result
-78	1.65	-28	'no found'
-45	-1.65	-16	'no found'

# Testing ServiciosHTTP

- Antes de empezar a escribir **test specs** para un service de APIs se necesita configurar un **módulo de testing** para poder aislar nuestros escenarios de tests y evitar así que se hagan peticiones http reales.
  - Hacer solicitudes reales retrasaría la ejecución de los tests
  - Ciertas API tienen límite de solicitudes; si se hacen llamadas en test se van a sobrepasar innecesariamente.
  - Se necesitan correr pruebas de forma offline
- Las peticiones se realizan contra un objeto simulado o **mock**. (Objetos que imitan el comportamiento de objetos reales pero de forma controlada).

# Testing ServiciosHTTP: Arrange

- Para configurar un módulo de testing se utiliza la clase **TestBed** de Angular.
- TestBed se utiliza para configurar e inicializar el entorno de la prueba. Genera un @NgModule de forma dinámica.

```
TestBed.configureTestingModule({}).
```

- En primer lugar, para la prueba es necesario establecer los *providers* de los servicios que se necesitan probar y los que se necesitan para hacer emulaciones.

```
import { TestBed, inject } from '@angular/core/testing';
import { UserService } from '../users.service';

describe('UserService', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [UserService]
    });
  });
});
```

# Testing ServiciosHTTP

- Angular provee una función para la inyección de dependencias en las pruebas: **inject()**
- La prueba que verifica si el servicio puede ser inyectado sin problemas la proporciona Angular:

```
it('should be created',  
  inject([UsersService], (service: UsersService) => {  
    expect(service).toBeTruthy();  
  }));
```



# Testing ServiciosHTTP

- En primer lugar se ha de proporcionar el contexto necesario para poder probar al servicioHTTP. Para ello importamos la clase **HttpClientTestingModule** que nos proporcionará un mock para realizar el test.

```
beforeEach(() => {  
  TestBed.configureTestingModule({  
    imports: [  
      HttpClientModule,  
    ],  
    providers: [  
      UserService  
    ],  
  });  
});
```

- En segundo lugar se han de proporcionar las dependencias que se utilizarán en las pruebas, como variables:

HttpTestingController  
es un mock de  
HttpClient

```
let service: UserService;  
let httpMock: HttpTestingController;  
  
beforeEach(() => {  
  TestBed.configureTestingModule({  
    imports: [  
      HttpClientModule,  
    ],  
    providers: [  
      UserService  
    ],  
  });  
  service = TestBed.get(UserService);  
  httpMock = TestBed.get(HttpTestingController);  
});
```

# Testing ServiciosHTTP

- Debemos planificar las tareas que esperamos que cumpla el método que queremos probar:
  - Debe hacer una solicitud al endpoint `http://jsonplaceholder.typicode.com/users`.
  - La solicitud debe ser por medio del método GET.
  - La respuesta que retorne debe ser igual a los mocks de datos que esperados.
  - Si estamos probando un caso de prueba en donde obtenemos una respuesta satisfactoria no se debería devolver ningún error.

# Testing ServiciosHTTP

- Escribiendo el caso de prueba [Arrange]:
  - Preparamos un mock de datos para la respuesta del endpoint, es decir esta sería la respuesta que normalmente esperamos que nos devuelva ese endpoint al hacer la petición.
  - Creamos dos variables *dataResponse* y *dataError* para guardar la respuesta de la solicitud y el error si éste se llegara a producir.

```
it('should return users', () => {  
  // Arrange  
  const mockResponse = {  
    results: [  
      {  
        'gender': 'male',  
        'name': {  
          'title': 'mr',  
          'first': 'samuel',  
          'last': 'ross'  
        },  
        'email': 'samuel.ross@example.com',  
      },  
    ],  
  };  
  let dataError, dataResponse;
```

# Testing ServiciosHTTP

- Escribiendo el caso de prueba [Act]:
  - Ejecutamos el método donde recibiremos la respuesta de la solicitud.
  - Inyectaremos como respuesta nuestro mocks de datos cuando se ejecute el endpoint

```
// Act|
service.getAllUsers()
  .subscribe((response) => {
    dataResponse = response['results'];
  }, (error) => {
    dataError = error;
  });
const req = httpMock.expectOne('https://randomuser.me/api/?results=25');
req.flush(mockResponse);
```

# Testing ServiciosHTTP

- Escribiendo el caso de prueba [Assert]:
  - Preparamos las verificaciones para comprobar que este método cumple con lo que esperamos:

```
// Assert
expect(dataResponse.length).toEqual(1);
expect(req.request.url).toEqual('https://randomuser.me/api/?results=25');
expect(req.request.method).toEqual('GET');
expect(dataError).toBeUndefined();
});
});
```

# Testing ServiciosHTTP

- Escribiendo el caso de prueba: Test del Error
  - Probamos qué pasa cuando ocurre un error:

```
it('should return an error', () => {  
  // Arrange  
  let dataError, dataResponse: any[];  
  // Act  
  service.getAllUsers()  
    .subscribe((response) => {  
    | dataResponse = response['results'];  
  }, (error) => {  
    | dataError = error;  
  });  
  httpMock.expectOne('https://randomuser.me/api/?results=25')  
    .error(new ErrorEvent('error'));  
  // Assert  
  expect(dataResponse).toBeUndefined();  
  expect(dataError).toBeDefined();  
});  
});
```

# Ejercicio 1

- Prepara las pruebas del método createUser.

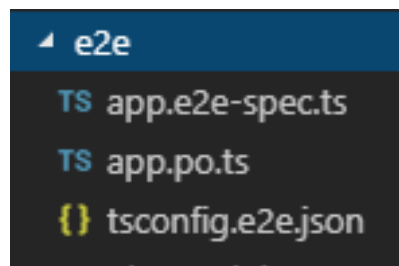
# Testing e2e

- El testing **End to End** es una metodología usada para hacer testing desde la perspectiva de usuario. El test asegura el funcionamiento de la aplicación según lo esperado, desde principio a fin.
- Conforme se realizan los tests se observa una interacción con el navegador igual que haría un usuario de la aplicación.
- En Angular los tests e2e se corren utilizando un framework denominado **Protractor**.
- Protractor utiliza el Webdriver de Selenium, API de automatización de navegadores y testing.
- Los tests se escriben mediante el framework de **Jasmine**



# Testing e2e

- Cuando se crea la aplicación de Angular se genera automáticamente un directorio con los dos archivos utilizados en las pruebas y uno de configuración general del proyecto:



- Los tests ejecutados con protractor dependen de los ficheros de pruebas (**spec.ts**) y del fichero de configuración de protractor (**protractor.conf.json**) situado en el *root* de la aplicación

# Testing e2e: protractor.conf.json

- Dentro del fichero, la línea **directConnect: true** permite a Protractor conectarse con los drivers de los navegadores soportados: Chrome, Firefox, Safari, and IE. Si fuera necesario correr los tests en un navegador diferente habría que configurar el servidor de Selenium

```
directConnect: false,  
baseUrl: 'http://localhost:4200/',  
seleniumAddress: 'http://localhost:4444/wd/hub',
```

# Configuración y ejecución de los tests

- Fichero **spec.ts** con las pruebas en si (sintaxis Jasmine)
- Fichero **po.ts: Las Page Objects** se basan en un patrón de diseño utilizado en automatización de tests para reducir la duplicación de código. Son clases que se utilizan como interfaces para los tests.
- Se deberían organizar los ficheros dentro de carpetas que representen los escenarios de prueba
  
- El test se ejecuta con la instrucción: **ng e2e**

# Preparando una prueba

- Preparamos el Page Objects de la página de inicio:
  - Hacemos el import de las clases que vamos a necesitar para trabajar con la vista:

```
1 import { browser, by, element } from 'protractor';
```

- Dentro de la clase preparamos los métodos que vamos a utilizar en el test:

```
export class AppPage {  
  navigateTo() {  
    return browser.get('/');  
  }  
  
  getParagraphText() {  
    return element(by.tagName ('a01-root h1')).getText();  
  }  
}
```

- Preparamos la prueba:

```
describe('angular01 App', () => {  
  let page: AppPage;  
  
  beforeEach(() => {  
    page = new AppPage();  
  });  
  
  it('should display Header message', () => {  
    page.navigateTo();  
    expect(page.getParagraphText()).toEqual('Cabecera de la aplicación');  
  });  
});
```

## Ejercicio 1

- Prepara las pruebas de la pantalla de inicio de una de las aplicaciones
- Referencia: [Angular End To End Testing](#)

# Referencias

[Angular End To End Testing](#)