Angular 8 Testing

https://angular.io/

Contenidos

- Herramientas de Desarrollo
- Introducción a las técnicas de pruebas
- EL proceso de pruebas
- Técnicas de pruebas
- Metodología
- Pruebas en Angular
  - Utilidades
  - Técnicas de pruebas y TDD
- Pruebas unitarias
  - Jasmine y Karma
  - Pruebas Unitarias Aisladas
  - Utilidades Angular para pruebas
- Pruebas E2E
  - Protractor
  - Selenium

HERRAMIENTAS DE DESARROLLO

IDEs

- Visual Studio Code - http://code.visualstudio.com/
  - VS Code is a Free, Lightweight Tool for Editing and Debugging Web Apps.
- StackBlitz - https://stackblitz.com
  - The online IDE for web applications. Powered by VS Code and GitHub.
- Angular IDE by Webclipse - https://www.genuitec.com/products/angular-ide
  - Built first and foremost for Angular. Turnkey setup for beginners; powerful for experts.
- IntelliJ IDEA - https://www.jetbrains.com/idea/
  - Capable and Ergonomic Java * IDE
- Webstorm - https://www.jetbrains.com/webstorm/
  - Lightweight yet powerful IDE, perfectly equipped for complex client-side development and server-side development with Node.js

Instalación de utilidades

Consideraciones previas

- Las utilidades son de línea de comandos.
- Para ejecutar los comandos es necesario abrir la consola comandos (Símbolo del sistema)
- Siempre que se realice una instalación o creación es conveniente “Ejecutar como Administrador” para evitar otros problemas.
- En algunos casos el firewall de Windows, la configuración del proxy y las aplicaciones antivirus pueden dar problemas.

GIT: Software de control de versiones

- Descargar e instalar: https://git-scm.com/
- Verificar desde consola de comandos:
  - git

Node.js: Entorno en tiempo de ejecución

- Descargar e instalar: https://nodejs.org
- Verificar desde consola de comandos:
  - node --version

npm: Node Package Manager

- Aunque se instala con el Node es conveniente actualizarlo:
  - npm update -g npm
- Verificar desde consola de comandos:
  - npm --version
- Configuración:
  - npm config edit
  - proxy=http://usr:pwd@proxy.dominion.com:8080 ← Símbolos: %HEX ASCII
- Generar fichero de dependencias package.json:
  - npm init
- Instalación de paquetes:
  - npm install -g grunt-cli grunt-init karma karma-cli ← Global (CLI)
  - npm install jasmine-core tslint --save --save-dev
  - npm install ← Dependencias en package.json
- Arranque del servidor:
  - npm start

Generación del esqueleto de aplicación

- Configurar un nuevo proyecto de Angular 2 es un proceso complicado y tedioso, con tareas como:
  - Crear la estructura básica de archivos y bootstrap
  - Configurar SystemJS o WebPack para transpilar el código
  - Crear scripts para ejecutar el servidor de desarrollo, tester, publicación, …
- Disponemos de diferentes opciones de asistencia:
  - Proyectos semilla (seed) disponibles en github
  - Generadores basados en Yeoman
  - Herramienta oficial de gestión de proyectos, Angular CLI.
- Angular CLI, creada por el equipo de Angular, es una *Command Line Interface* que permite generar proyectos y plantillas de código desde consola, así como ejecutar un servidor de desarrollo o lanzar los tests de la aplicación. (https://cli.angular.io/)
  - npm install -g @angular/cli@latest

Proyectos semilla

- Proyectos semilla:
  - https://github.com/angular/quickstart
  - http://mgechev.github.io/angular2-seed
  - https://github.com/ghpabs/angular2-seed-project
  - https://github.com/cureon/angular2-sass-gulp-boilerplate
  - https://angularclass.github.io/angular2-webpack-starter
  - https://github.com/LuxDie/angular2-seed-jade
  - https://github.com/justindujardin/angular2-seed
- Generadores de código basados en Yeoman (http://yeoman.io/generators):
  - https://github.com/FountainJS/generator-fountain-angular2
  - https://github.com/ericmdantas/generator-ng-fullstack
- NOTA: Es conveniente verificar si están actualizados a la última versión de Angular 2, pueden estar desactualizados.

Creación y puesta en marcha

- Acceso al listado de comandos y opciones
  - $ ng help
- Nuevo proyecto
  - $ ng new myApp
  - Esto creará la carpeta myApp con un esqueleto de proyecto ya montado según la última versión de Angular y todo el trabajo sucio de configuración de WebPack para transpilar el código y generar un bundle, configuración de tests, lint y typescript, etc.
  - Además, instala todas las dependencias necesarias de npm e incluso inicializa el proyecto como un repositorio de GIT.
- Servidor de desarrollo
  - ng serve
  - Esto lanza tu app en la URL http://localhost:4200 y actualiza el contenido cada vez que guardas algún cambio.

Generar código

- Hay elementos de código que mantienen una cierta estructura y no necesitas escribirla cada vez que creas un archivo nuevo.
- El comando generate permite generar algunos de los elementos habituales en Angular.
  - Componentes: ng generate component my-new-component
  - Directivas: ng g directive my-new-directive
  - Pipe: ng g pipe my-new-pipe
  - Servicios: ng g service my-new-service
  - Clases: ng g class my-new-class
  - Interface: ng g interface my-new-interface
  - Enumerados: ng g enum my-new-enum
  - Módulos: ng g module my-module
  - Guardian (rutas): ng generate guard my-guard

  --module my-module Indica el modulo si no es el principal
  --project my-lib Indica el proyecto si no es el principal

Schematics en Angular CLI (v.6)

- Schematics es una herramienta de flujo de trabajo para la web moderna; permite aplicar transformaciones al proyecto, como crear un nuevo componente, actualizar el código para corregir cambios importantes en una dependencia, agregar una nueva opción o marco de configuración a un proyecto existente.
- Generadores de componentes de arranque Angular Material:
  - Un componente de inicio que incluye una barra de herramientas con el nombre de la aplicación y la navegación lateral:
    - ng generate @angular/material:material-nav --name=my-nav
  - Un componente de panel de inicio que contenga una lista de tarjetas de cuadrícula dinámica:
    - ng generate @angular/material:material-dashboard --name=my-dashboard
  - Un componente de tabla de datos de inicio que está pre configurado con un datasource para ordenar y paginar:
    - ng generate @angular/material:material-table --name=my-table

Nuevos comandos (v.6)

- **ng update <package>** analiza el package.json de la aplicación actual y utiliza la heurísticas de Angular para recomendar y realizar las actualizaciones que necesita la aplicación.
  - npm install -g @angular/cli
  - npm install @angular/cli
  - ng update @angular/cli
- **ng add <package>** permite agregar nuevas capacidades al proyecto y puede actualizar el proyecto con cambios de configuración, agregar dependencias adicionales o estructurar el código de inicialización específico del paquete.
  - ng add @angular/pwa - Convierte la aplicación en un PWA agregando un manifiesto de aplicación y un service worker.
  - ng add @ng-bootstrap/schematics - Agrega ng-bootstrap a la aplicación.
  - ng add @angular/material - Instala y configura Angular Material y el estilo, y registrar nuevos componentes de inicio en ng generate.
  - ng add @angular/elements - Agrega el polyfill document-register-element.js y las dependencias necesarios para los Angular Elements.

Pruebas

- Son imprescindibles en entornos de calidad: permite ejecutar las pruebas unitarias, las pruebas de extremo a extremo o comprobar la sintaxis.
- Para comprobar la sintaxis: Puedes ejecutar el analizador con el comando:
  - ng lint
- Para ejecutar tests unitarios: Puedes lanzar los tests unitarios con karma con el comando:
  - ng test
- Para ejecutar tests e2e: Puedes lanzar los tests end to end con protractor con el comando:
  - ng e2e

Despliegue

- Construye la aplicación en la carpeta /dist
  - ng build
  - ng build --dev
- Paso a producción, construye optimizándolo todo para producción
  - ng build --prod
  - ng build --prod --env=prod
  - ng build --target=production --environment=prod
- Precompila la aplicación
  - ng build --prod --aot

Estructura de directorios de soporte

- src
- e2e
  - src
    - app.e2e-spec.ts
    - app.po.ts
  - protractor.conf.js
  - tsconfig.json
- .editorconfig
- .gitignore
- angular.json
- browserslist
- karma.conf.js
- package.json
- protractor.conf.js
- README.md
- tsconfig.json
- tsconfig.app.json
- tsconfig.spec.json
- tslint.json

- src: Fuentes de la aplicación
- e2e: Test end to end
- Ficheros de configuración de librerías y herramientas
- Posteriormente aparecerán los siguientes directorios:
  - node_modules: Librerías y herramientas descargadas
  - dist: Resultado para publicar en el servidor web
  - coverage: Informes de cobertura de código
- Tsconfig.xxxx.json: Configuración del compilador TypeScript para la aplicación y las pruebas.

Estructura de directorios de aplicación

- app
  - app.component.css
  - app.component.html
  - app.component.spec.ts
  - app.component.ts
  - app.module.ts
  - index.ts
- assets
- environments
  - environment.prod.ts
  - environment.ts
- favicon.ico
- index.html
- main.ts
- polyfills.ts
- styles.css
- test.ts

- app: Carpeta que contiene los ficheros fuente principales de la aplicación.
- assets: Carpeta con los recursos que se copiaran a la carpeta build.
- environments: configuración de los diferentes entornos.
- main.ts: Arranque del módulo principal de la aplicación. No es necesario modificarle.
- favicon.ico: Icono para el navegador.
- index.html: Página principal (SPA).
- style.css: Se editará para incluir CSS global de la web, se concatena, minimiza y enlaza en index.html automáticamente.
- test.ts: pruebas del arranque.
- polyfills.js: importación de los diferentes módulos de compatibilidad ES6 y ES7.

Webpack

- Webpack (https://webpack.github.io/) es un empaquetador de módulos, es decir, permite generar un archivo único con todos aquellos módulos que necesita la aplicación para funcionar.
- Toma módulos con dependencias y genera archivos estáticos correspondientes a dichos módulos.
- Webpack va mas allá y se ha convertido en una herramienta muy versátil. Entre otras cosas, destaca que:
  - Puede generar solo aquellos fragmentos de JS que realmente necesita cada página.
    - Dividir el árbol de dependencias en trozos cargados bajo demanda
    - Haciendo más rápida la carga inicial
  - Tiene varios loaders para importar y empaquetar también otros recursos (CSS, templates, …) así como otros lenguajes (ES6 con Babel, TypeScript, SaSS, etc).
  - Sus plugins permiten hacer otras tareas importantes como por ejemplo minimizar y ofuscar el código.

GIT

- Preséntate a Git
  - git config --*global* user.name "Your Name Here“
  - git config --*global* user.email your_email@youremail.com
- Crea un repositorio central
  - https://github.com/
- Conecta con el repositorio remoto
  - git remote add origin https://github.com/username/myproject.git
  - git push -u origin master
- Actualiza el repositorio con los cambios:
  - git commit -m "first commit"
  - git push
- Para clonar el repositorio:
  - git clone https://github.com/username/myproject.git *local-dir*
- Para obtener las últimas modificaciones:
  - git pull

INTRODUCCIÓN A LAS TÉCNICAS DE PRUEBAS

Introducción

- El software generado en la fase de implementación no puede ser "entregado" al cliente para que lo utilice, sin practicarle antes una serie de pruebas.
- La fase de pruebas tienen como objetivo encontrar defectos en el sistema final debido a la omisión o mala interpretación de alguna parte del análisis o el diseño. Los defectos deberán entonces detectarse y corregirse en esta fase del proyecto.
- En ocasiones los defectos pueden deberse a errores en la implementación de código (errores propios del lenguaje o sistema de implementación), aunque en esta etapa es posible realizar una efectiva detección de los mismos, estos deben ser detectados y corregidos en la fase de implementación.
- La prueba puede ser llevada a cabo durante la implementación, para verificar que el software se comporta como su diseñador pretendía, y después de que la implementación esté completa.

Introducción

- Esta fase tardía de prueba comprueba la conformidad con los requerimientos y asegura la fiabilidad del sistema.
- Las distintas clases de prueba utilizan **clases de datos de prueba** diferentes:
  - La **prueba estadística**.
  - La **prueba de defectos**.
  - La **prueba de regresión**.

Prueba estadística

- La **prueba estadística** se puede utilizar para probar el rendimiento del programa y su confiabilidad.
- Las pruebas se diseñan para reflejar la frecuencia de entradas reales de usuario.
- Después de ejecutar las pruebas, se puede hacer una estimación de la confiabilidad operacional del sistema.
- El rendimiento del programa se puede juzgar midiendo la ejecución de las pruebas estadísticas.

Prueba de defectos

- La **prueba de defectos** intenta incluir áreas donde el programa no está de acuerdo con sus especificaciones.
- Las pruebas se diseñan para revelar la presencia de defectos en el sistema.
- Cuando se han encontrado defectos en un programa, éstos deben ser localizados y eliminados. A este proceso se le denomina **depuración**.
- La prueba de defectos y la depuración son consideradas a veces como parte del mismo proceso. En realidad, son muy diferentes, puesto que la prueba establece la existencia de errores, mientras que la depuración se refiere a la localización y corrección de estos errores.

Prueba de regresión

- Durante la depuración se debe generar hipótesis sobre el comportamiento observable del programa para probar entonces estas hipótesis esperando provocar un fallo y encontrar el defecto que causó la anomalía en la salida.
- Después de descubrir un error en el programa, debe corregirse y volver a probar el sistema.
- A esta forma de prueba se le denomina **prueba de regresión**.
- La prueba de regresión se utiliza para comprobar que los cambios hechos a un programa no han generado nuevos fallos en el sistema.

Conflicto de intereses

- Aparte de esto, en cualquier proyecto software existe un **conflicto de intereses** inherente que aparece cuando comienza la prueba:
  - Desde un punto de vista psicológico, el análisis, diseño y codificación del software son tareas **constructivas**.
    - El ingeniero de software crea algo de lo que está orgulloso, y se enfrenta a cualquiera que intente sacarle defectos.
    - La prueba, entonces, puede aparecer como un intento de “romper" lo que ha construido el ingeniero de software.
  - Desde el punto de vista del constructor, la prueba se puede considerar (psicológicamente) **destructiva**.
    - Por tanto, el constructor anda con cuidado, diseñando y ejecutando pruebas que demuestren que el programa funciona, en lugar de detectar errores.
    - Desgraciadamente los errores seguirán estando, y si el ingeniero de software no los encuentra, lo hará el cliente.

Desarrollador como probador

- A menudo, existen ciertos **malentendidos** que se pueden deducir equivocadamente de la anterior discusión:
  1. El desarrollador del software no debe entrar en el proceso de prueba.
  2. El software debe ser “puesto a salvo” de extraños que puedan probarlo de forma despiadada.
  3. Los encargados de la prueba sólo aparecen en el proyecto cuando comienzan las etapas de prueba.
- Cada una de estas frases es incorrecta.
- El **desarrollador** siempre es responsable de probar las unidades individuales (módulos) del programa, asegurándose de que cada una lleva a cabo la función para la que fue diseñada.
- En muchos casos, también se encargará de la prueba de integración de todos los elementos en la estructura total del sistema.

Grupo independiente de prueba

- Sólo una vez que la arquitectura del software esté completa, entra en juego un **grupo independiente de prueba**, que debe eliminar los problemas inherentes asociados con el hecho de permitir al constructor que pruebe lo que ha construido. Una prueba independiente elimina el conflicto de intereses que, de otro modo, estaría presente.
- En cualquier caso, el desarrollador y el grupo independiente deben trabajar estrechamente a lo largo del proyecto de software para asegurar que se realizan pruebas exhaustivas.
- Mientras se dirige la prueba, el desarrollador debe estar disponible para corregir los errores que se van descubriendo.

Principios fundamentales

- Hay 5 principios fundamentales respecto a las metodologías de pruebas que deben quedar claros desde el primer momento aunque volveremos a ellos continuamente:
  - Las pruebas exhaustivas no son viables
  - Ejecución de pruebas bajo diferentes condiciones
  - El proceso de pruebas no puede demostrar la ausencia de defectos
  - Las pruebas no garantizan ni mejoran la calidad del software
  - Inicio temprano de pruebas

Las pruebas exhaustivas no son viables

- Es imposible, inviable, crear casos de prueba que cubran todas las posibles combinaciones de entrada y salida que pueden llegar a tener las funcionalidades (salvo que sean triviales).
- Por otro lado, en proyectos cuyo número de casos de uso o historias de usuario desarrollados sea considerable, se requeriría de una inversión muy alta en cuanto a recursos y tiempo necesarios para cubrir con pruebas todas las funcionalidades del sistema.
- Por lo tanto es conveniente realizar un análisis de riesgos de todas las funcionalidades y determinar en este punto cuales serán objeto de prueba y cuales no, creando pruebas que cubran el mayor número de casos de prueba posibles.

Ejecución de pruebas bajo diferentes condiciones

- El plan de pruebas determina la condiciones y el número de ciclos de prueba que se ejecutarán sobre las funcionalidades del negocio.
- Por cada ciclo de prueba, se generan diferentes tipos de condiciones, basados principalmente en la variabilidad de los datos de entrada y en los conjuntos de datos utilizados.
- No es conveniente, ejecutar en cada ciclo, los casos de prueba basados en los mismos datos del ciclo anterior, dado que con mucha probabilidad, se obtendrán los mismos resultados.
- Ejecutar ciclos bajo diferentes tipos de condiciones, permitirá identificar posibles fallos en el sistema que antes no se detectaron y no son fácilmente reproducibles.

El proceso no puede demostrar la ausencia de defectos

- Independientemente de la rigurosidad con la que se haya planeado el proceso de pruebas de un producto, nunca será posible garantizar al ejecutar este proceso, la ausencia total de defectos (es inviable una cobertura del 100%).
- Una prueba se considera un éxito si detecta un error. Si no detecta un error no significa que no haya error, significa que no se ha detectado.
- Un proceso de pruebas riguroso puede garantizar una reducción significativa de los posibles fallos y/o defectos del software, pero nunca podrá garantizar que el software no fallará en producción.

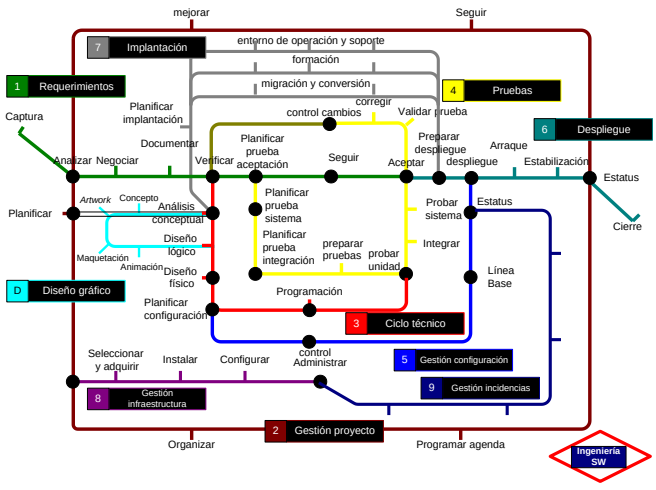
Las pruebas no garantizan ni mejoran la calidad del software

- Las pruebas ayudan a **mejorar la percepción** de la calidad permitiendo la eliminación de los defectos detectados.
- La calidad del software viene determinada por las metodologías y buenas practicas empleadas en el desarrollo del mismo.
- Las pruebas **permiten medir la calidad** del software, lo que permite, a su vez, mejorar los procesos de desarrollo que son los que conllevan la mejora de la calidad y permiten garantizar un nivel determinado de calidad.

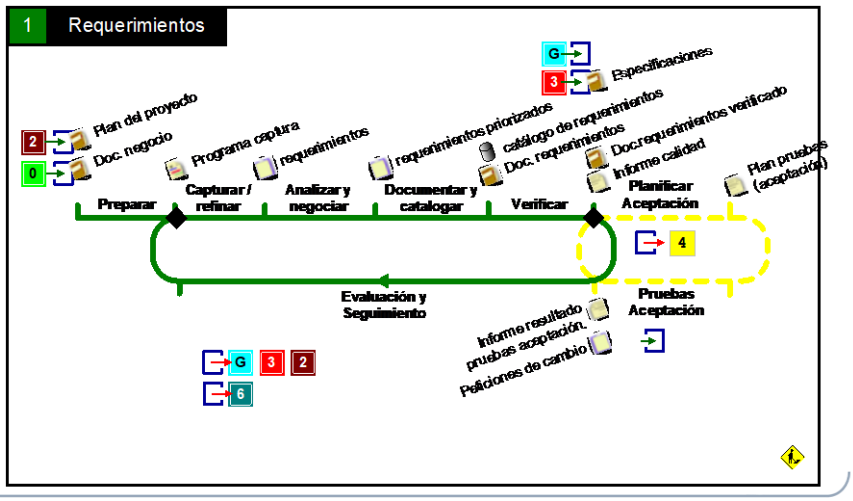
Inicio temprano de pruebas

- Si bien la fase de pruebas es la última del ciclo de vida, las actividades del proceso de pruebas deben ser incorporadas desde la fase de especificación, incluso antes de que se ejecutan las etapas de análisis y diseño.
- De esta forma los documentos de especificación y de diseño deben ser sometidos a revisiones y validaciones, lo que ayudará a detectar problemas en la lógica del negocio mucho antes de que se escriba una sola línea de código.
- Cuanto mas temprano se detecte un defecto, ya sea sobre los entregables de especificación, diseño o sobre el producto, menor impacto tendrá en el desarrollo y menor será el costo dar solución a dichos defectos.

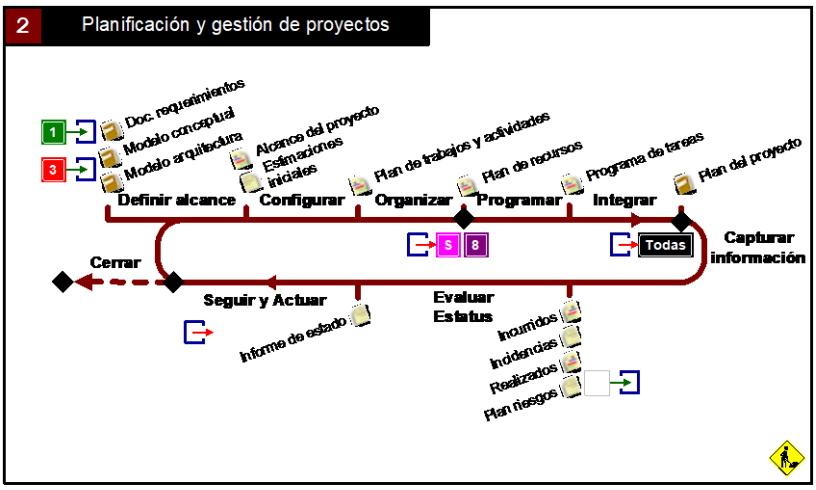
Integración en el ciclo de vida
Símil del mapa del Metro



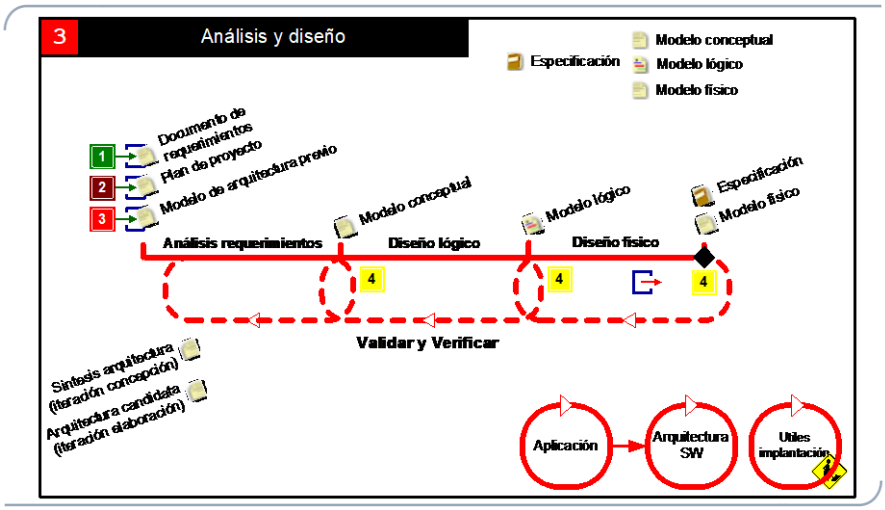
Línea 1: Requerimientos



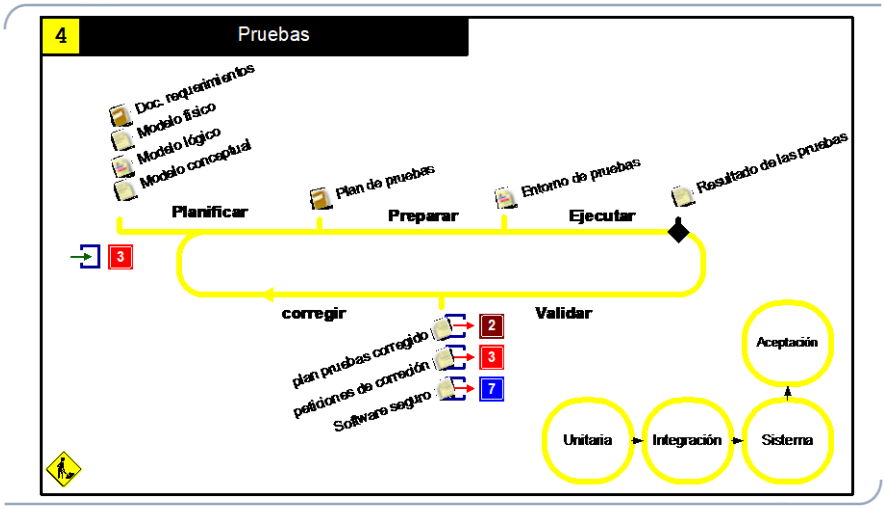
Línea 2: Planificación y gestión



Línea 3: Análisis y diseño



Línea 4: Pruebas

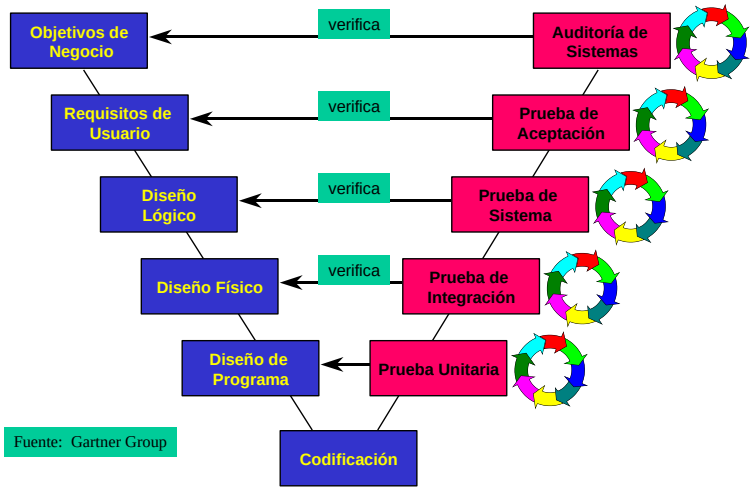


EL PROCESO DE PRUEBAS

El proceso

- Excepto para programas pequeños, los sistemas no deberían probarse como un único elemento indivisible.
- Los sistemas grandes se construyen a partir de subsistemas que se construyen a partir de módulos, compuestos de funciones y procedimientos.
- El proceso de prueba debería trabajar por etapas, llevando a cabo la prueba de forma incremental a la vez que se realiza la implementación del sistema, siguiendo el modelo en V.

Proceso de pruebas: Ciclo en V



Ciclo en V

- El modelo en V establece una simetría entre las fases de desarrollo y las pruebas.
- Las principales consideraciones se basan en la inclusión de las actividades de planificación y ejecución de pruebas como parte del proyecto de desarrollo.
- Inicialmente, la ingeniería del sistema define el papel del software y conduce al análisis de los requisitos del software, donde se establece el campo de información, la función, el comportamiento, el rendimiento, las restricciones y los criterios de validación del software. Al movernos hacia abajo, llegamos al diseño y, por último, a la codificación, el vértice de la V.
- Para desarrollar las pruebas seguimos el camino ascendente por la otra rama de la V.
- Partiendo de los elementos más básicos, probamos que funcionan como deben (lo que hacen, lo hacen bien). Los combinamos y probamos que siguen funcionando como deben. Para terminar probamos que hacen lo que deben (que hacen todo lo que tienen que hacer).

Niveles de pruebas

- **Pruebas Unitarias**: verifican la funcionalidad y estructura de cada componente individualmente, una vez que ha sido codificado.
- **Pruebas de Integración**: verifican el correcto ensamblaje entre los distintos componentes una vez que han sido probados unitariamente, con el fin de comprobar que interactúan correctamente a través de sus interfaces, cubren la funcionalidad establecida y se ajustan a los requisitos no funcionales especificados en las verificaciones correspondientes.
- **Pruebas de Regresión**: verifican que los cambios sobre un componente de un sistema de información no introducen un comportamiento no deseado o errores adicionales en otros componentes no modificados.
- **Pruebas del Sistema**: ejercitan profundamente el sistema comprobando la integración del sistema de información globalmente, verificando el funcionamiento correcto de las interfaces entre los distintos subsistemas que lo componen y con el resto de sistemas de información con los que se comunica.
- **Pruebas de Aceptación**: validan que un sistema cumple con el funcionamiento esperado y permitir al usuario de dicho sistema, que determine su aceptación desde el punto de vista de su funcionalidad y rendimiento.

Pruebas Unitarias

- Las pruebas unitarias tienen como objetivo verificar la funcionalidad y estructura de cada componente individualmente, una vez que ha sido codificado.
- Con las pruebas unitarias verificas el diseño de los programas, vigilando que no se producen errores y que el resultado de los programas es el esperado.
- Estas pruebas las efectúa normalmente la misma persona que codifica o modifica el componente y que, también normalmente, genera un juego de ensayo para probar y depurar las condiciones de prueba.
- Las pruebas unitarias constituyen la prueba inicial de un sistema y las demás pruebas deben apoyarse sobre ellas.

Pruebas Unitarias

- Existen dos **enfoques** principales para el diseño de casos de prueba:
  - **Enfoque estructural o de caja blanca**. Se verifica la estructura interna del componente con independencia de la funcionalidad establecida para el mismo.

    Por tanto, no se comprueba la corrección de los resultados, sólo si éstos se producen. Ejemplos de este tipo de pruebas pueden ser ejecutar todas las instrucciones del programa, localizar código no usado, comprobar los caminos lógicos del programa, etc.
  - **Enfoque funcional o de caja negra**. Se comprueba el correcto funcionamiento de los componentes del sistema de información, analizando las entradas y salidas y verificando que el resultado es el esperado. Se consideran exclusivamente las entradas y salidas del sistema sin preocuparse por la estructura interna del mismo.
- El enfoque que suele adoptarse para una prueba unitaria está claramente orientado al diseño de casos de caja blanca, aunque se complemente con caja negra.

Pruebas Unitarias

- Los **pasos necesarios** para llevar a cabo las pruebas unitarias son los siguientes:
  - **Ejecutar todos los casos de prueba** asociados a cada verificación establecida en el plan de pruebas, registrando su resultado. Los casos de prueba deben contemplar tanto las condiciones válidas y esperadas como las inválidas e inesperadas.
  - **Corregir los errores o defectos encontrados y repetir las pruebas que los detectaron**. Si se considera necesario, debido a su implicación o importancia, se repetirán otros casos de prueba ya realizados con anterioridad.

Pruebas Unitarias

- La prueba unitaria se da por finalizada cuando se hayan realizado todas las verificaciones establecidas y no se encuentre ningún defecto, o bien se determine su suspensión.
- Al finalizar las pruebas, obtienes las **métricas de calidad del componente** y las contrastas con las existentes antes de la modificación:
  - Número ciclomático.
  - Cobertura de código.
  - Porcentaje de comentarios.
  - Defectos hallados contra especificaciones o estándares.
  - Rendimientos.

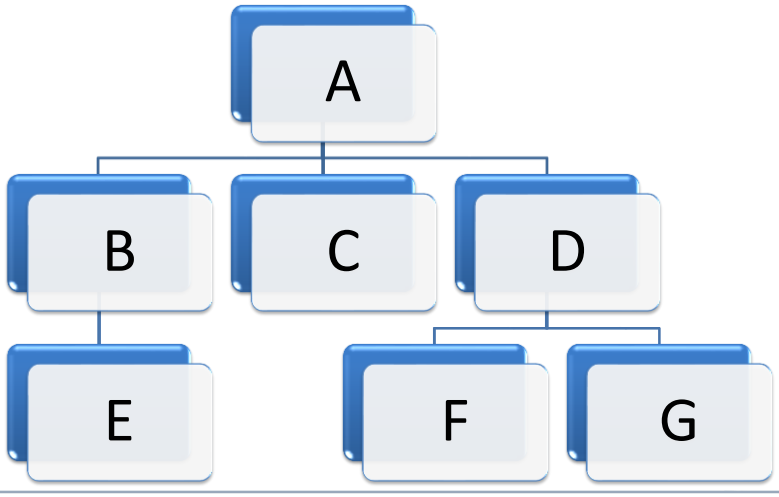
Pruebas de Integración

- Las pruebas de integración te permiten verificar el correcto ensamblaje entre los distintos componentes una vez que han sido probados unitariamente, con el fin de comprobar que interactúan correctamente a través de sus interfaces, tanto internas como externas, cubren la funcionalidad establecida y se ajustan a los requisitos no funcionales especificados en las verificaciones correspondientes.
- Se trata de probar los caminos lógicos del flujo de los datos y mensajes a través de un conjunto de componentes relacionados que definen una cierta funcionalidad.
- En las pruebas de integración examinas las interfaces entre grupos de componentes o subsistemas para asegurar que son llamados cuando es necesario y que los datos o mensajes que se transmiten son los requeridos.
- Debido a que en las pruebas unitarias es necesario crear módulos auxiliares que simulen las acciones de los componentes invocados por el que se está probando, y a que se han de crear componentes "conductores" para establecer las precondiciones necesarias, llamar al componente objeto de la prueba y examinar los resultados de la prueba, a menudo se combinan los tipos de prueba unitarias y de integración.

Pruebas de Integración

- Los **tipos fundamentales de integración** son los siguientes:
  - **Integración incremental:** combinas el siguiente componente que debes probar con el conjunto de componentes que ya están probados y vas incrementando progresivamente el número de componentes a probar.
  - **Integración no incremental:** pruebas cada componente por separado y, luego, los integras todos de una vez realizando las pruebas pertinentes. Este tipo de integración se denomina también Big-Bang (gran explosión).
- Con el tipo de prueba incremental lo más probable es que los problemas te surjan al incorporar un nuevo componente o un grupo de componentes al previamente probado. Los problemas serán debidos a este último o a las interfaces entre éste y los otros componentes.

Pruebas de Integración



Estrategias de integración

- **De arriba a abajo (top-down):** el primer componente que se desarrolla y prueba es el primero de la jerarquía (A).
  - Los componentes de nivel más bajo se sustituyen por componentes auxiliares o resguardos, para simular a los componentes invocados. En este caso no son necesarios componentes conductores.
  - Una de las ventajas de aplicar esta estrategia es que las interfaces entre los distintos componentes se prueban en una fase temprana y con frecuencia.
- **De abajo a arriba (bottom-up):** en este caso se crean primero los componentes de más bajo nivel (E, F, G) y se crean componentes conductores para simular a los componentes que los llaman.
  - A continuación se desarrollan los componentes de más alto nivel (B, C, D) y se prueban. Por último dichos componentes se combinan con el que los llama (A). Los componentes auxiliares son necesarios en raras ocasiones.
  - Este tipo de enfoque permite un desarrollo más en paralelo que el enfoque de arriba a abajo, pero presenta mayores dificultades a la hora de planificar y de gestionar.
- **Estrategias combinadas:** A menudo es útil aplicar las estrategias anteriores conjuntamente. De este modo, se desarrollan partes del sistema con un enfoque "top-down", mientras que los componentes más críticos en el nivel más bajo se desarrollan siguiendo un enfoque "bottom-up".
  - En este caso es necesaria una planificación cuidadosa y coordinada de modo que los componentes individuales se "encuentren" en el centro.

Pruebas de Regresión

- El objetivo de las pruebas de regresión es eliminar el efecto onda, es decir, comprobar que los cambios sobre un componente de un sistema de información, no introducen un comportamiento no deseado o errores adicionales en otros componentes no modificados.
- Las pruebas de regresión se deben llevar a cabo cada vez que se hace un cambio en el sistema, tanto para corregir un error como para realizar una mejora.
- No es suficiente probar sólo los componentes modificados o añadidos, o las funciones que en ellos se realizan, sino que también es necesario controlar que las modificaciones no produzcan efectos negativos sobre el mismo u otros componentes.
- Normalmente, este tipo de pruebas implica la repetición de las pruebas que ya se han realizado previamente, con el fin de asegurar que no se introducen errores que puedan comprometer el funcionamiento de otros componentes que no han sido modificados y confirmar que el sistema funciona correctamente una vez realizados los cambios.

Pruebas de Regresión

- Las pruebas de regresión **pueden incluir**:
  - La repetición de los casos de pruebas que se han realizado anteriormente y están directamente relacionados con la parte del sistema modificada.
  - La revisión de los procedimientos manuales preparados antes del cambio, para asegurar que permanecen correctamente.
  - La obtención impresa del diccionario de datos de forma que se compruebe que los elementos de datos que han sufrido algún cambio son correctos.
- El **responsable** de realizar las pruebas de regresión será el equipo de desarrollo junto al técnico de mantenimiento, quién a su vez, será responsable de especificar el plan de pruebas de regresión y de evaluar los resultados de dichas pruebas.

Pruebas del Sistema

- Las pruebas del sistema tienen como objetivo ejercitar profundamente el sistema comprobando la integración del sistema de información globalmente, verificando el funcionamiento correcto de las interfaces entre los distintos subsistemas que lo componen y con el resto de sistemas de información con los que se comunica.
- Son pruebas de integración del sistema de información completo, y permiten probar el sistema en su conjunto y con otros sistemas con los que se relaciona para verificar que las especificaciones funcionales y técnicas se cumplen. Dan una visión muy similar a su comportamiento en el entorno de producción.
- Una vez que se han probado los componentes individuales y se han integrado, se prueba el sistema de forma global. En esta etapa pueden distinguirse diferentes tipos de pruebas, cada uno con un objetivo claramente diferenciado.

Pruebas del Sistema

- **Pruebas funcionales**: dirigidas a asegurar que el sistema de información realiza correctamente todas las funciones que se han detallado en las especificaciones dadas por el usuario del sistema.
- **Pruebas de humo**: son un conjunto de pruebas aplicadas a cada nueva versión, su objetivo es validar que las funcionalidades básicas de la versión se cumplen según lo especificado. Impiden la ejecución el plan de pruebas si detectan grandes inestabilidades o si elementos clave faltan o son defectuosos.
- **Pruebas de comunicaciones**: determinan que las interfaces entre los componentes del sistema funcionan adecuadamente, tanto a través de dispositivos remotos, como locales. Asimismo, se han de probar las interfaces hombre-máquina.
- **Pruebas de rendimiento**: consisten en determinar que los tiempos de respuesta están dentro de los intervalos establecidos en las especificaciones del sistema.
- **Pruebas de volumen**: consisten en examinar el funcionamiento del sistema cuando está trabajando con grandes volúmenes de datos, simulando las cargas de trabajo esperadas.
- **Pruebas de sobrecarga o estrés**: consisten en comprobar el funcionamiento del sistema en el umbral límite de los recursos, sometiéndole a cargas masivas. El objetivo es establecer los puntos extremos en los cuales el sistema empieza a operar por debajo de los requisitos establecidos.

Pruebas del Sistema

- **Pruebas de disponibilidad de datos**: consisten en demostrar que el sistema puede recuperarse ante fallos, tanto de equipo físico como lógico, sin comprometer la integridad de los datos.
- **Pruebas de configuración**: consisten en comprobar todos y cada uno de los dispositivos, en sus propiedades mínimo y máximo posibles.
- **Pruebas de usabilidad**: consisten en comprobar la adaptabilidad del sistema a las necesidades de los usuarios, tanto para asegurar que se acomoda a su modo habitual de trabajo, como para determinar las facilidades que aporta al introducir datos en el sistema y obtener los resultados.
- **Pruebas extremo a extremo (e2e)**: consisten en interactuar con la aplicación como un usuario regular lo haría, cliente-servidor, y evaluando las respuestas para el comportamiento esperado.
- **Pruebas de operación**: consisten en comprobar la correcta implementación de los procedimientos de operación, incluyendo la planificación y control de trabajos, arranque y re-arranque del sistema, etc.
- **Pruebas de entorno**: consisten en verificar las interacciones del sistema con otros sistemas dentro del mismo entorno.
- **Pruebas de seguridad**: consisten en verificar los mecanismos de control de acceso al sistema para evitar alteraciones indebidas en los datos.

Pruebas de Aceptación

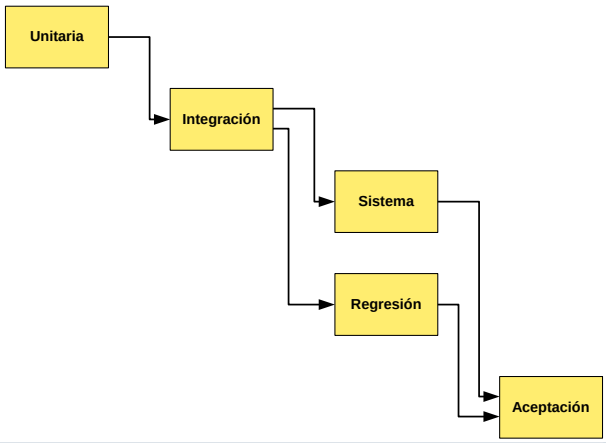
- El objetivo de las pruebas de aceptación es validar que un sistema cumple con el funcionamiento esperado y permitir al usuario de dicho sistema, que determine su aceptación desde el punto de vista de su funcionalidad y rendimiento.
- Las pruebas de aceptación son preparadas por el usuario del sistema y el equipo de desarrollo, aunque la ejecución y aprobación final corresponde al usuario.
- Estas pruebas van dirigidas a comprobar que el sistema cumple los requisitos de funcionamiento esperado recogidos en el catálogo de requisitos y en los criterios de aceptación del sistema de información, y conseguir la aceptación final del sistema por parte del usuario.

Pruebas de Aceptación

- Previamente a la realización de las pruebas, el responsable de usuarios revisa los criterios de aceptación que se especificaron previamente en el plan de pruebas del sistema y dirige las pruebas de aceptación final.
- La validación del sistema se consigue mediante la realización de pruebas de caja negra que demuestran la conformidad con los requisitos y que se recogen en el plan de pruebas, el cual define las verificaciones a realizar y los casos de prueba asociados.
- Dicho plan está diseñado para asegurar que se satisfacen todos los requisitos funcionales especificados por el usuario teniendo en cuenta, a su vez, los requisitos no funcionales relacionados con el rendimiento, seguridad de acceso al sistema, a los datos y procesos, así como a los distintos recursos del sistema.
- La formalidad de estas pruebas dependerá en mayor o menor medida de cada organización, y vendrá dada por la criticidad del sistema, el número de usuarios implicados en las mismas y el tiempo del que se disponga para llevarlas cabo, entre otros.

Niveles de pruebas y orden de ejecución.

- De tal forma que la secuencia de pruebas es:



Tipos de Pruebas

- Las actividades de las pruebas pueden centrarse en comprobar el sistema en base a un objetivo o motivo especifico:
  - Una función a realizar por el software.
  - Una característica no funcional como el rendimiento o la fiabilidad.
  - La estructura o arquitectura del sistema o el software.
  - Los cambios para confirmar que se han solucionado los defectos o localizar los no intencionados.
- Las pruebas se pueden clasificar como:
  - Pruebas funcionales
  - Pruebas no funcionales
  - Pruebas estructurales
  - Pruebas de mantenimiento

TÉCNICAS

Introducción

- Durante las fases anteriores de definición y de desarrollo, has intentado construir el software partiendo de un concepto abstracto y llegando a una implementación tangible.
- A continuación llega la prueba, debes crear una serie de casos de prueba que intenten demoler el software que ha sido construido. De hecho, la prueba es uno de los pasos de la ingeniería del software que, por lo menos psicológicamente, se puede ver como destructivo en lugar de constructivo.
- Para comenzar vamos a establecer una serie de reglas que sirven como objetivos de prueba:
  1. La prueba es un proceso de ejecución de un programa con la intención de descubrir un error.
  2. Un buen caso de prueba es aquel que tiene una alta probabilidad de mostrar un error no descubierto hasta entonces.
  3. Una prueba tiene éxito si descubre un error no detectado hasta entonces.

Introducción

- Los objetivos anteriores suponen un cambio dramático de punto de vista. Nos quitan la idea que, normalmente, tenemos de que una prueba tiene éxito si no descubre errores.
- El objetivo es diseñar pruebas que sistemáticamente saquen a la luz diferentes clases de errores, haciéndolo con la menor cantidad de tiempo y esfuerzo.
- Si la prueba se lleva a cabo con éxito (de acuerdo con el objetivo anteriormente establecido), descubrirá errores en el software.
- Como ventaja secundaria, la prueba demuestra hasta qué punto las funciones del software parecen funcionar de acuerdo con las especificaciones y parecen alcanzarse los requisitos de rendimiento. Además, los datos que se van recogiendo a medida que se lleva a cabo la prueba proporcionan una buena indicación de la fiabilidad del software y, de alguna manera, indican la calidad del software como un todo.
- Sin embargo, hay una cosa que no puede hacer la prueba: "**La prueba no puede asegurar la ausencia de defectos, sólo puede demostrar que existen defectos en el software**". Es importante tener en mente esta frase pesimista mientras se lleva a cabo la prueba.

Casos de prueba

- El diseño de pruebas para el software puede requerir tanto esfuerzo como el propio diseño inicial del producto.
- Recordando el objetivo de la prueba, debes diseñar pruebas que tengan la mayor probabilidad de encontrar el mayor número de errores con la mínima cantidad de esfuerzo y de tiempo.
- Un **caso de prueba** especifica una forma de probar el sistema, incluyendo la entrada con la que se ha de probar, los resultados que se esperan obtener y las condiciones bajo las que ha de probarse.

Casos de prueba

- Cualquier producto de ingeniería puede ser probado de una de estas dos formas:
  - **Conociendo la función específica** para la que fue diseñado el producto, se pueden llevar a cabo pruebas que demuestren que cada función es completamente operativa.
  - **Conociendo el funcionamiento del producto**, se pueden desarrollar pruebas que aseguren que "todas las piezas encajan"; o sea, que la operación interna se ajusta a las especificaciones y que todos los componentes internos se han comprobado de forma adecuada.
- La primera forma, se denomina **prueba de caja negra** y la segunda **prueba de caja blanca**.

Casos de prueba

- La **prueba de caja negra** se refiere a las pruebas que se llevan a cabo sobre la interfaz del software.
- O sea, los casos de prueba pretenden demostrar que las funciones del software son operativas, que la entrada se acepta de forma adecuada y que se produce una salida correcta, así como que la integridad de la información externa se mantiene.
- Una prueba de caja negra examina algunos aspectos del modelo fundamental del sistema sin tener demasiado en cuenta la estructura lógica interna del software, siendo las **más adecuadas para probar los casos de uso del "modelo de casos de uso"**. Las pruebas de caja negra pueden ser ejecutadas por personal sin experiencia en ingeniería de software.

Casos de prueba

- La **prueba de caja blanca** del software se basa en el minucioso examen de los detalles procedimentales.
- Se comprueban los caminos lógicos del software proponiendo casos de prueba que ejerciten conjuntos específicos de condiciones y/o bucles. Se puede examinar el estado del programa en varios puntos para determinar si el estado real coincide con el esperado o afirmado.
- A primera vista parecería que una prueba de caja blanca muy profunda nos llevaría a tener "programas 100% correctos". Todo lo que tienes que hacer es definir todos los caminos lógicos, desarrollar casos de prueba que los ejerciten y evaluar los resultados; es decir, generar casos de prueba que ejerciten exhaustivamente la lógica del programa.

Casos de prueba

- Desgraciadamente, la prueba exhaustiva presenta problemas. Incluso para pequeños programas, el número de caminos lógicos posibles puede ser enorme. La prueba exhaustiva es imposible para los grandes sistemas software. **Las pruebas de caja blanca están especialmente indicadas para validar las realizaciones de casos de uso del "modelo de diseño"**.
- La prueba de caja blanca, sin embargo, no se debe desechar como impracticable.
- Se puede elegir y ejercitar una serie de caminos lógicos importantes. Se pueden comprobar las estructuras de datos más importantes para ver su validez. Se pueden combinar los atributos de la prueba de caja blanca así con los de caja negra, para llegar a un método que valide la interfaz del software y asegure selectivamente que el funcionamiento interno del software es correcto.

Casos de prueba

- Las pruebas de caja blanca y caja negra enfocan las pruebas desde el punto de vista de la ejecución.
- Existe otro punto de vista, el del usuario, donde las pruebas deben ir enfocadas a buscar los errores producidos en **la interfaz con el usuario**.
- Al contrario de las pruebas que exigen la ejecución de software, las pruebas dinámicas, **las pruebas estáticas** se basan en el examen manual (revisiones) y en el análisis automatizado (análisis estático) del código o de cualquier otra documentación del proyecto sin ejecutar el código.
- Formalmente, los casos de prueba escritos consisten principalmente en tres secciones:
  - Identificación
  - Definición
  - Resultados

Identificación

- **Identificador**: Es un identificador único para futuras referencias, por ejemplo, mientras se describe un defecto encontrado.
- **Nombre**: El caso de prueba debe tener un título entendible por las personas, para facilitar la comprensión del propósito del caso de prueba y su campo de aplicación.
- **Propósito**: Contiene una breve descripción del propósito de la prueba y la funcionalidad probada.
- **Creador**: Es el nombre del analista o diseñador de pruebas, quien ha desarrollado pruebas o es responsable de su desarrollo.
- **Versión**: Número de la definición actual del caso de prueba.
- **Referencias**: Identificadores de requerimientos, casos de uso o de especificaciones funcionales que están cubiertos por el caso de prueba.
- **Dependencias**: Orden de ejecución de casos de pruebas, razón de las dependencias.

Definición

- **Precondiciones**: situación previa a la ejecución de pruebas o características de un objeto de pruebas antes de llevar a la práctica (ejecución) un caso de prueba.
- **Valores de entrada**: descripción de los datos de entrada de un objeto de pruebas.
- **Resultados esperados**: datos de salida que se espera que produzca un objeto de prueba.
- **Poscondiciones**: características de un objeto de prueba tras la ejecución de pruebas, descripción de su situación tras la ejecución de las pruebas.
- **Requisitos**: características del objeto de pruebas que el caso de prueba debe evaluar. Contiene información acerca de la configuración del hardware o software en el cuál se ejecutará el caso de prueba.
- **Acciones**: Pasos a realizar para completar la prueba.

Resultados

- **Salida obtenida**: Contiene una breve descripción de lo que el analista encuentra después de que los pasos de prueba se hayan completado.
- **Resultado**: Indica el resultado cualitativo de la ejecución del caso de prueba, a menudo con un Correcto/Fallido.
- **Severidad**: Indica el impacto del defecto en el sistema: Critico, Grave, Normal, Menor.
- **Evidencia**: En los casos que se aplica, contiene información donde se evidencia la salida obtenida y como reproducirla. Puede ir acompañado por un enlace, un pantallazo (screenshot), una consulta a base de datos, el contenido de un fichero de trazas …
- **Seguimiento**: Si un caso de prueba falla, frecuentemente la referencia al defecto implicado se debe enumerar. Contiene el código correlativo del defecto, a menudo corresponde al código del sistema de seguimiento de defectos que se esté usando.
- **Estado**: Indica si el caso de prueba está: No iniciado, En curso o Terminado.

Pruebas de interfaces de usuario

- A diferencia de los casos anteriores, las pruebas de interfaces de usuario se basan en la experiencia para descubrir los casos de prueba.
- La técnica se basa en una serie de cuestionarios que contienen las situaciones más habituales de error. La identificación de los casos de prueba la realizas recorriendo cada una de las cuestiones de cada lista, estudiando si es o no es aplicable. En caso de ser aplicable necesitas diseñar un caso de prueba.
- Las listas que te proponemos son más o menos estándar. No son cerradas. Debes irlas completando sobre la base de tu experiencia.

Ventanas

- ❑ ¿Se abrirán las ventanas basándose en órdenes basadas en el teclado o en un menú?
- ❑ ¿Se puede ajustar el tamaño, mover y desplegar la ventana?
- ❑ ¿Está todo el contenido de la información dentro de la ventana accesible adecuadamente con el ratón, teclas de función, flechas de dirección y teclado?
- ❑ ¿Se genera adecuadamente cuando se sobrescribe y se vuelve a abrir?
- ❑ ¿Están operativas todas las funciones relacionadas con la ventana?
- ❑ ¿Están disponibles y desplegados apropiadamente en la ventana todos los menús emergentes, barras de herramientas, barras deslizantes, cuadros de diálogo, botones, iconos y otros controles importantes?
- ❑ Cuándo se despliegan varias ventanas, ¿se representa adecuadamente el nombre de cada ventana?
- ❑ ¿Está resaltada adecuadamente la ventana activa?
- ❑ Si se utiliza multitarea, ¿están actualizadas todas las ventanas en los momentos adecuados?
- ❑ ¿Causan las selecciones múltiples o incorrectas del ratón dentro de la ventana efectos secundarios inesperados?
- ❑ ¿Están de acuerdo con las especificaciones los indicadores de audio y/o de color de la ventana o como consecuencia de operaciones de la ventana?
- ❑ ¿Se cierra adecuadamente la ventana?

Menús emergentes y operaciones con el ratón

- ❑ ¿Se muestra la barra de menú apropiada en el contexto apropiado?
- ❑ ¿Despliega la barra de menú de la aplicación características relacionadas con el sistema (por ejemplo: la pantalla de un reloj)?
- ❑ ¿Funcionan adecuadamente las operaciones de despliegue?
- ❑ ¿Funcionan adecuadamente los menús de escape, paletas y barras de herramientas?
- ❑ ¿Están listadas adecuadamente todas las funciones del menú y subfunciones emergentes?
- ❑ ¿Son todas las funciones del menú accesibles con el ratón?
- ❑ ¿Es correcto el tipo, tamaño y formato de texto?
- ❑ ¿Es posible invocar todas las funciones del menú usando su orden alternativa de texto?

Menús emergentes y operaciones con el ratón (cont.)

- ❑ ¿Están resaltadas las funciones del menú (o difuminadas) dependiendo del contexto de las operaciones actuales de la ventana?
- ❑ ¿Se ejecutan todas las funciones de cada menú como se anunciaba?
- ❑ ¿Son suficientemente claros los nombres de las funciones del menú?
- ❑ ¿Hay ayuda disponible para cada elemento del menú y es sensible al contexto?
- ❑ ¿Se reconocen apropiadamente las operaciones del ratón a lo largo de todo el contexto interactivo?
- ❑ Si se necesitan múltiples clic, ¿están apropiadamente reconocidos en el contexto?
- ❑ Si el ratón tiene varios botones, ¿son reconocidos apropiadamente en el contexto?
- ❑ ¿Cambian adecuadamente el cursor, el indicador de procesamiento y el puntero al invocar diferentes operaciones?

Entrada de datos

- ❑ ¿Se repiten y son introducidos adecuadamente los datos alfanuméricos en el sistema?
- ❑ ¿Funcionan adecuadamente los modos gráficos de entrada de datos (por ejemplo: una barra deslizante)?
- ❑ ¿Se reconocen adecuadamente los datos no válidos?
- ❑ ¿Son inteligibles los mensajes de entrada de datos?

Documentación y ayuda

- ❑ ¿Describe con exactitud la documentación cómo conseguir cada modo de empleo?
- ❑ ¿Es exacta la descripción de cada secuencia de interacción?
- ❑ ¿Son exactos los ejemplos?
- ❑ ¿Son consistentes con el programa real la terminología, las descripciones del menú y las respuestas del sistema?
- ❑ ¿Es relativamente fácil localizar ayuda en la documentación?
- ❑ ¿Se pueden solucionar problemas fácilmente con la documentación?
- ❑ ¿Son exactos y completos la tabla de contenido y el índice?
- ❑ ¿Facilita el diseño del documento (distribución, tipos de letra, indentación, grafos) la comprensión y rápida asimilación de la información?
- ❑ ¿Están descritos con gran detalle los mensajes de error para el usuario en el documento?
- ❑ Si se utilizan enlaces de hipertexto, ¿son exactos y completos?

Técnicas estáticas

- Al contrario que las pruebas dinámicas, que exigen la ejecución de software, las técnicas de pruebas estáticas se basan en el examen manual (revisiones) y en el análisis automatizado (análisis estático) del código o de cualquier otra documentación del proyecto sin ejecutar el código.
- Las revisiones constituyen una forma de probar los productos de trabajo del software (incluyendo el código) y pueden realizarse antes de ejecutar las pruebas dinámicas. Los defectos detectados durante las revisiones al principio del ciclo (por ejemplo, los defectos encontrados en los requisitos) a menudo son mucho más baratos de eliminar que los detectados durante las pruebas realizadas ejecutando el código.
- Una revisión podría hacerse íntegramente como una actividad manual, pero también existen herramientas de soporte.

METODOLOGÍA

Introducción

- Durante las fases anteriores se han dedicado a construir el sistema, partiendo de las especificaciones han realizado el análisis, el diseño y la implementación.
- En la fase de pruebas te dedicas a destruir el sistema. El objetivo es identificar el mayor número de posibles causas de fallo y probar si se producen errores.
- En la fase de pruebas **estudiaremos toda la documentación anteriormente generada**: el modelo de casos de uso con el documento de requisitos adicionales, el modelo de análisis, el modelo de diseño, el modelo de implementación y el documento de descripción de arquitectura.
- La calidad de dicha documentación es fundamental para la fase de pruebas, incidiendo directamente en la calidad del proceso de pruebas.

Introducción

- Una baja calidad en la documentación o la ausencia de algunos documentos degrada la validez del proceso de pruebas llegando incluso a imposibilitar dicho proceso.
- El **modelo de pruebas** describe cómo se prueban los componentes ejecutables del modelo de implementación con pruebas de integración y de sistema, así como aspectos específicos como puede ser la consistencia de la interfaz de usuario, si el manual del usuario cumple su cometido, y otros.
- A la fase de prueba se llega después de completar la fase de implementación, es cometido de dicha fase haber realizado las pruebas de unidad.

Elementos

- Como vimos en su momento, la base de la fase de pruebas son los **casos de prueba.** Recuerda que un caso de prueba específica una forma de probar el sistema, incluyendo la entrada con la que se ha de probar, los resultados que se esperan obtener y las condiciones bajo las que ha de probarse.
- Un **procedimiento de prueba** especifica cómo realizar uno o varios casos de prueba o partes de éstos.
- En el procedimiento documentas los pasos que deben darse para cada uno de los casos de prueba. Los procedimientos de prueba pueden reutilizarse para varios casos de prueba similares. Así mismo, un caso de prueba puede estar incluido en varios procedimientos de prueba.

Elementos

- Un **componente de prueba** automatiza uno o varios procedimientos de prueba o partes de éstos.
- Los componentes de prueba se diseñan e implementan de forma específica para proporcionar las entradas, controlar la ejecución e informar de la salida de los elementos a probar.
- Los componentes de prueba son necesarios puesto que la mayoría de las veces el elemento que estas probando no dispone de un interfaz de usuario que te permita la comunicación directa con él.

Elementos

- Un **defecto** es una anomalía del sistema, como por ejemplo un síntoma de un fallo software o un problema descubierto en una revisión. Los defectos deben estar documentados indicando síntomas, posibles causas y consecuencias.
- Existen **herramientas específicas de prueba de software** que permiten establecer los procedimientos de pruebas, ejecutar los procedimientos y evaluar los resultados guardando un registro de los mismos. Si utilizas dichas herramientas, en muchos casos, evitarás la necesidad de desarrollar componentes de prueba.
- Por lo tanto, el modelo de prueba está compuesto de casos de prueba, procedimientos de prueba y componentes de prueba. El modelo se organiza mediante los planes de prueba.

Plan de pruebas

- La prueba de sistemas es cara.
- La creciente inclusión del software como un elemento más de muchos sistemas productivos y la importancia de los "costes" asociados a un fallo del mismo están motivando la creación de pruebas minuciosas y bien planificadas.
- No es raro que una organización de desarrollo de software gaste el 40 por 100 del esfuerzo total de un proyecto en la prueba.
- En casos extremos, la prueba del software para actividades críticas (por ejemplo, control de tráfico aéreo, o control de reactores nucleares) puede costar ¡de 3 a 5 veces más que el resto de los pasos de la ingeniería del software juntos!

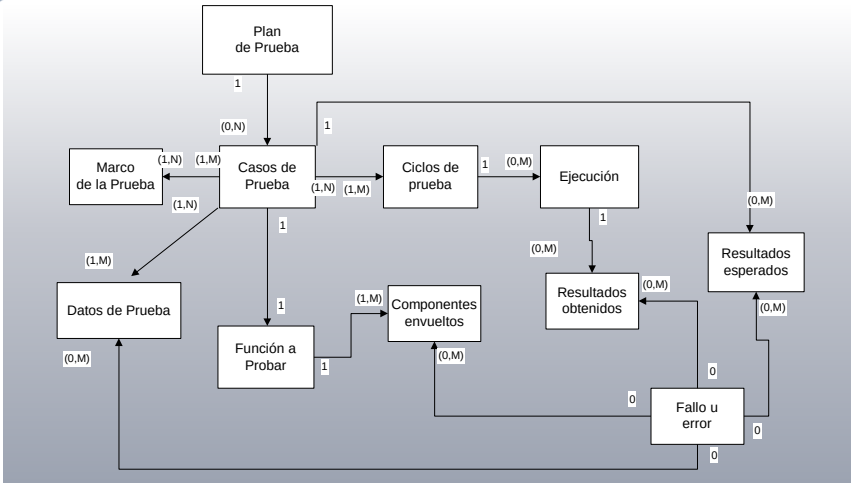
Plan de pruebas

- Se necesita una planificación cuidadosa para obtener lo máximo del proceso de prueba y controlar los costes.
- El propósito del plan de pruebas es explicitar el alcance, enfoque, recursos requeridos, calendario, responsables y manejo de riesgos de un proceso de pruebas.
- Puede haber un plan global que explicite el énfasis a realizar sobre los distintos tipos de pruebas (verificación, integración).

Arquitectura del plan de pruebas

- Casos de prueba (qué probar).
  - Elemento a probar.
  - Condiciones a probar.
  - Juegos de pruebas que aplican.
  - Resultados esperados.
- Procedimientos de prueba (cómo probar).
- Calendario de pruebas (ciclos de prueba).
- Recursos.
- Componentes de pruebas.
  - Modelo de pruebas.
  - Drivers y stubs.
  - Guiones de prueba (scripts).
- Entorno de pruebas.
  - Repositorio y herramientas de prueba.

Arquitectura conceptual del plan de pruebas



Detalles del Plan de pruebas

Identificador del plan

- Preferiblemente de alguna forma mnemónica que permita relacionarlo con su alcance.
- A modo de ejemplo: TP-Global (plan global del proceso de pruebas), TP-REQ-Seguridad1 (plan de verificación del requerimiento 1 de seguridad), TP-Contr-X (plan de verificación del contrato asociado al evento de sistema X), TP-Unit-Despachador.iniciar (plan de prueba unitario para el método iniciar de la clase Despachador).
- Como todo artefacto del desarrollo, está sujeto a control de configuración, por lo que debe distinguirse adicionalmente la versión y fecha del plan.

Detalles del Plan de pruebas

Alcance

- Indicas el nivel de prueba y las propiedades / elementos del software a ser probado.

Elementos a probar

- Indicas la configuración a probar y las condiciones mínimas que debe cumplir para comenzar a aplicarle el plan.
- Por un lado, es difícil y arriesgado probar una configuración que aún contiene fallos; por otro lado, si esperamos a que todos los módulos estén perfectos, puede que detectemos fallos graves demasiado tarde.

Detalles del Plan de pruebas

Estrategia

- Describes la técnica, patrón y/o herramientas a utilizar en el diseño de los casos de prueba.
- Por ejemplo, en el caso de pruebas unitarias de un procedimiento, esta sección podría indicar: "Se aplicará la estrategia caja negra de valores límite de la precondición" o "Ejercicio de los caminos básicos válidos".
- En lo posible, la estrategia debe precisar el número mínimo de casos de prueba a diseñar, por ejemplo el 100% de las fronteras, el 60% de los caminos ciclomáticos,... La estrategia también especifica el grado de automatización que se exigirá, tanto para la generación de casos de prueba como para su ejecución.

Detalles del Plan de pruebas

Procedimientos de prueba

- Especificas las condiciones bajo las cuales, el plan debe ser:
  - Suspendido.
  - Repetido.
  - Finalizado.
- En algunas circunstancias (que deben ser especificadas) el proceso de prueba debe suspenderse a la vista de los defectos o fallos que se han detectado. Al corregirse los defectos, el proceso de prueba previsto por el plan puede continuar, pero debes especificar a partir de qué punto, ya que puede ser necesario repetir algunas pruebas.
- Los criterios de culminación pueden ser tan simples como aprobar el número mínimo de casos de prueba diseñados o tan complejos como tomar en cuenta no sólo el número mínimo, sino también el tiempo previsto para las pruebas y la tasa de detección de fallos.

Detalles del Plan de pruebas

Tangibles

- Especificas los documentos a entregar al terminar el proceso previsto por el plan (por ejemplo: subplanes, especificación de pruebas, casos de prueba, resumen gerencial del proceso e histórico de pruebas).
- No es suficiente con ejecutar simplemente las pruebas, sino que se deben almacenar sus resultados de forma sistemática. Es posible auditar el proceso para comprobar que se ha llevado a cabo de forma correcta.

Procedimientos especiales

- Identificas el grafo de las tareas necesarias para preparar y ejecutar las pruebas, así como cualquier habilidad especial que se requiere.

Detalles del Plan de pruebas

Infraestructura y entorno

- Especificas las propiedades necesarias y deseables del ambiente de prueba, incluyendo las características del hardware, el software de sistemas, cualquier otro software necesario para llevar a cabo las pruebas, así como la colocación específica del software a probar (por ejemplo, qué módulos se colocan en qué máquinas de una red local) y la configuración del software de apoyo.

Recursos Humanos

- La sección incluye una estimación de los recursos humanos necesarios para el proceso.
- También se indica cualquier requerimiento especial del proceso: actualización de licencias, espacio de oficina, tiempo en la máquina de producción, seguridad.

Detalles del Plan de pruebas

Calendario

- Esta sección describe los hitos del proceso de prueba y el grafo de dependencia en el tiempo de las tareas a realizar.
- Estableces un calendario de prueba global y la asignación de recursos para este calendario. Esto, obviamente, está ligado al calendario general de desarrollo del proyecto.

Gestión de riesgos

- Especifica los riesgos del plan, las acciones mitigantes y de contingencia.

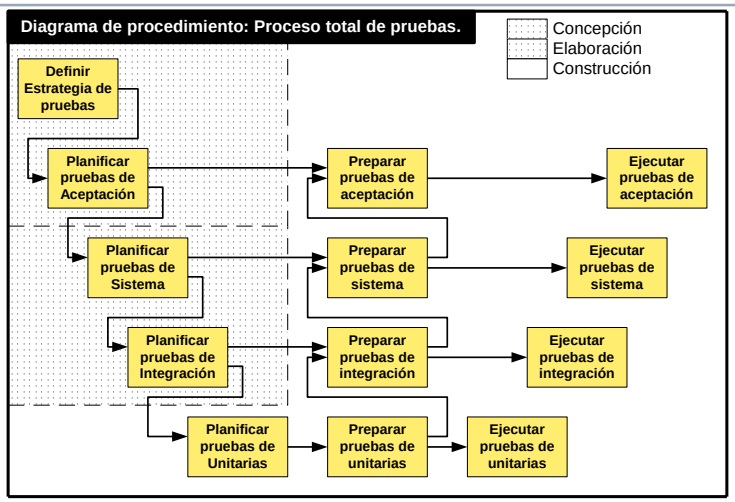
Responsables

- Especifica quién es el responsable de cada una de las tareas previstas en el plan.

Características del plan de pruebas

- Este plan debería contemplar cantidades significativas de eventualidades, de forma que errores en el diseño o la implementación se puedan solucionar y parte del personal que realiza las pruebas se pueda dedicar a otras actividades.
- Como otros planes, el plan de prueba no es un documento estático. Debe revisarse regularmente puesto que la prueba es una actividad dependiente del avance de la implementación. Si parte del sistema a probar está incompleto, el proceso de prueba del sistema no puede comenzar.
- Debes contar con un plan de pruebas global y tantos planes de pruebas como sean necesarios para cubrir el alcance del plan global.
- La planificación de las pruebas comienza desde las fases iniciales del desarrollo tal y como muestra el siguiente diagrama:

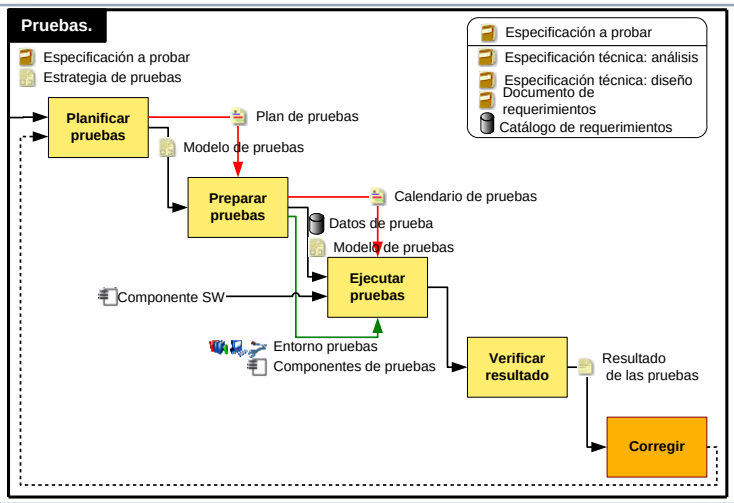
Proceso total de pruebas



Metodología

- El primer paso que debes dar es la **planificación de la prueba**, pasando a continuación a su **diseño**.
- Para diseñar la prueba debes identificar y diseñar los casos de prueba de integración, los casos de prueba de sistema, los casos de prueba de regresión y los procedimientos de prueba.
- Una vez concluido el diseño, debes **implementar los componentes de pruebas necesarios**.
- Por último realizas las **pruebas de integración** y la **prueba de sistema**.
- Concluyes el proceso con la **evaluación de la prueba**.

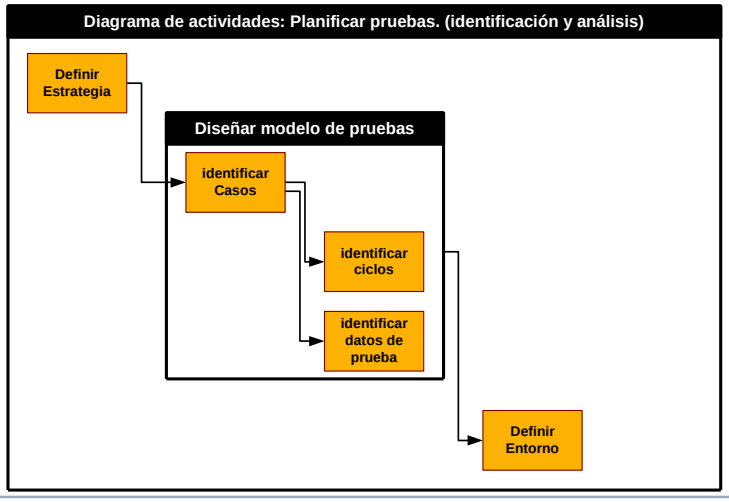
Metodología



Planificar la prueba

- El primer paso que debes dar es planificar los esfuerzos de prueba describiendo una estrategia y estimando los requisitos de la prueba, tanto en recursos humanos como en sistemas necesarios.
- El modelo de casos de uso y los requisitos adicionales, junto con el modelo de diseño te ayudan a decidir el tipo adecuado de pruebas y a estimar el esfuerzo necesario para llevarlas a cabo.
- Utilizando ambos modelos estableces una estrategia de prueba decidiendo qué tipo de pruebas ejecutar, cómo y cuándo ejecutar dichas pruebas y cómo determinar si el esfuerzo de la prueba tiene éxito.

Planificar las pruebas



Diseñar la prueba

- Para diseñar la prueba empiezas por identificar y describir los casos de prueba de cada componente.
- La selección de las técnicas de pruebas depende factores adicionales como pueden ser requisitos contractuales o normativos, documentación disponible, tiempo, presupuesto, conocimientos, experiencia, …
- Cuando dispongas de los casos de prueba, identificas y estructuras los procedimientos de prueba describiendo cómo ejecutar los casos de prueba.

Diseño de los casos de prueba unitarias

- Los casos de prueba unitaria deben estar integrados en el proceso de codificación.
- Es responsabilidad del programador crear e implementar los casos de prueba que verifiquen que el componente cumple con los requerimientos de implementación recibido. Habitualmente se emplean técnicas de caja blanca con un determinado nivel de cobertura de código.
- En caso de que debas crear casos de prueba adicionales a los del programador, se recomienda utilizar técnicas de caja negra como técnica complementaria a la empleada por el programador.
- El desarrollo guiado por pruebas o Test-driven development (TDD) es una práctica de ingeniería de software que involucra escribir las pruebas primero y refactorizar depues. Para escribir las pruebas generalmente se utilizan las pruebas unitarias:
  - Antes de crear el componente, se escribe una prueba y se verifica que las pruebas fallan.
  - A continuación, se implementa el componente con el código que hace que la prueba pase satisfactoriamente y se verifica.
  - Seguidamente se refactoriza el código escrito con el código real del componente y se verifica.

Bases de las pruebas unitarias

Entradas

- Requerimiento del componente
- Diseño de detalle
- Código

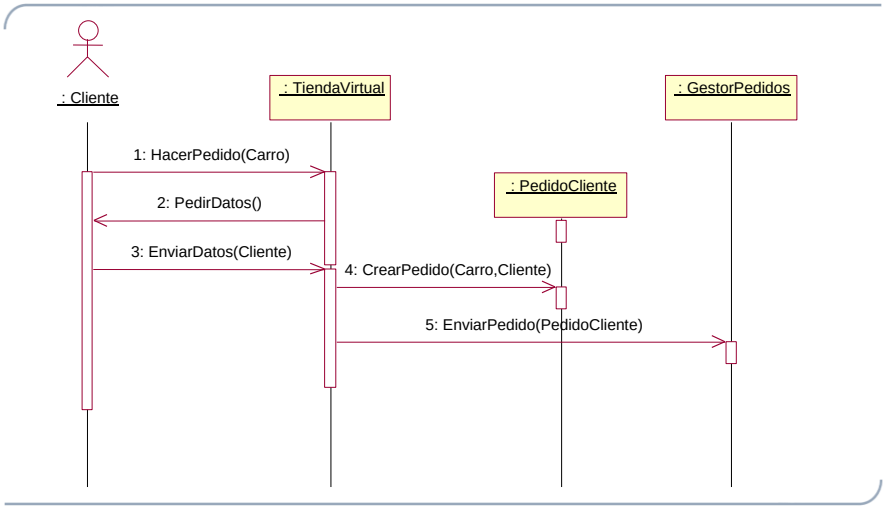
Objetivos

- Componentes
- Programas
- Módulos de bases de datos

Diseño de los casos de prueba de integración

- Los casos de prueba de integración se utilizan para verificar que los componentes interaccionan entre sí de la forma apropiada después de haber sido integrados en una construcción.
- En este caso te serán de especial utilidad los diagramas de secuencia, los diagramas de colaboración y los diagramas de estados que se elaboran en las realizaciones de casos de uso del "*modelo de diseño*".
- Dichos diagramas representan las interacciones entre los objetos del diseño.
- Para encontrar los casos de prueba debes buscar las combinaciones de entrada, salida y estado que den lugar a escenarios interesantes que utilicen los objetos representados en los diagramas.
- El conjunto de casos de prueba que selecciones debe ir encaminado a conseguir los objetivos del plan de pruebas con el esfuerzo mínimo.
- Para ello eliges casos de uso que no se solapen entre sí y que cada uno de ellos pruebe un camino o escenario interesante en la realización de caso de uso.

Diagrama de secuencias



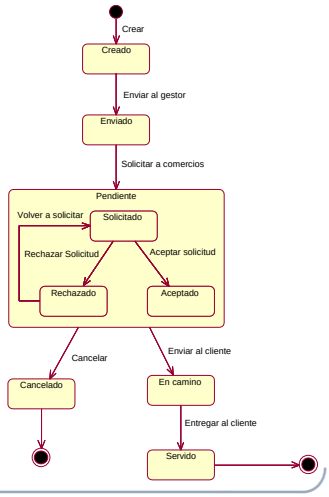
Análisis del Diagrama de secuencias

- Puedes extraer un caso de pruebas que describa cómo probar la secuencia representada en el diagrama, tomando en cuenta el estado inicial del sistema, las entradas del actor y demás circunstancias necesarias que hacen que ocurra la secuencia.
- Si el diagrama representara varias secuencias posibles, debes extraer tantos casos de pruebas como sean necesarios para probar todas las secuencias.

S1	Los datos del carro son correctos	SI	SI	NO	NO
S3	Los datos del cliente son correctos	SI	NO	SI	NO
S2	Pedir datos de cliente	X	X		
S4	Crear pedido	X			
S5	Enviar pedido	X			

Diagramas de estado

- De igual forma debes actuar con los diagramas de estado que representan todos los posibles estados de un elemento del sistema y sus correspondientes transiciones de estado.
- Debes extraer casos de prueba que te permitan probar cada una de las transiciones con sus posibles itinerarios.
- No olvides que NO debes centrarte exclusivamente en las transiciones posibles sino que también debes estudiar las imposibles (quizás las más importantes), como por ejemplo la cancelación de un pedido que se encuentra en camino.



Bases de las pruebas de integración

Entradas

- Diseño de software y sistema
- Arquitectura
- Flujos de trabajo
- Casos de uso

Objetivos

- Componentes
- Interfaces
- Infraestructura
- Bases de datos
- Configuración del sistema

Diseño de los casos de prueba de sistema

- Las pruebas de sistema se usan para probar que el sistema funciona correctamente como un todo.
- Las pruebas de sistema se centran fundamentalmente en probar combinaciones de diferentes casos de uso ejecutados bajo diferentes condiciones.
- Las condiciones deben incluir, a su vez, combinaciones diferentes de configuraciones hardware, niveles de carga del sistema, número de actores y tamaños de base de datos.
- La forma de identificar los casos de prueba depende estrechamente del tipo de sistema que estés probando y de sus características especiales.

Diseño de los casos de prueba de sistema

- El **modelo de casos de uso** te servirá de punto de partida, fíjate en sus flujos y requisitos especiales (como son las restricciones funcionales y los requisitos de rendimiento).
- El **modelo de despliegue**, elaborado en la fase de diseño, también te aporta mucha información.
- El modelo de despliegue describe la distribución física del sistema, asignando funcionalidades a nodos de cómputo. Así mismo, describe las relaciones de los nodos entre sí mediante medios de comunicación (Internet, Intranet y similares).

Diseño de los casos de prueba de sistema

- Debes hacer hincapié en los casos de uso que:
  - Son críticos para el sistema.
  - Tienen tiempos de respuesta determinados.
  - Manejan grandes volúmenes de información.
  - Usan frecuentemente los recursos del sistema (procesadores, bases de datos,...).
  - Utilizan medios de comunicación.
  - Requieren procesamiento especial (en paralelo, multiprocesador,...)

Bases de las pruebas de sistema

Entradas

- Casos de uso
- Especificaciones funcionales
- Especificaciones de requisitos de software y sistema
- Checklist:
  - Usabilidad, accesibilidad, …
- Informe de análisis de riesgos

Objetivos

- Sistema
- Manuales de sistema, usuario y funcionamiento
- Bases de datos
- Configuración del sistema

Diseño de los casos de prueba de regresión

- Las pruebas de regresión son aquellas que diseñan o se repiten para buscar los errores que se producen después del proceso de corrección de los errores.
- Algunos de los casos de pruebas diseñados en los pasos anteriores son utilizados en las pruebas de regresión.
- Dependiendo de la cobertura, la repetición de casos de pruebas existentes y se deben diseñar casos específicos buscando el impacto del cambio.
- El diseño de los casos de prueba utilizados en las pruebas de regresión lo debes realizar con suficiente flexibilidad como para que soporten los cambios del software que estás probando.

Bases de las pruebas de regresión

Entradas

- Notificaciones de defectos
- Solicitudes de cambio
- Casos de uso
- Especificaciones funcionales
- Especificaciones de requisitos de software y sistema
- Informe de análisis de riesgos

Objetivos

- Componentes
- Sistema
- Manuales de sistema, usuario y funcionamiento
- Bases de datos
- Configuración del sistema

Diseño de los casos de prueba de aceptación

- Las pruebas de aceptación se basan fundamentalmente en el punto de vista del usuario final.
- El **modelo de casos de uso y las historias de usuario** te indican como va a utilizar el usuario el sistema y te permite crear casos de prueba para cada uso del sistema.
- Requieren una revisión formal de que toda la funcionalidad pactada es entregada, su objetivo principal no es tanto localizar defectos como evaluar la buena disposición del sistema para su despliegue y explotación.
- Las pruebas de aceptación tienen una vertiente contractual y normativa que toman como base los criterios de aceptación previstos en el contrato de desarrollo o las normas de obligado cumplimiento.
- En dicha situación, los casos de prueba se diseñan para comprobar el efectivo cumplimiento de cada uno de los criterios y normas.

Bases de las pruebas de aceptación

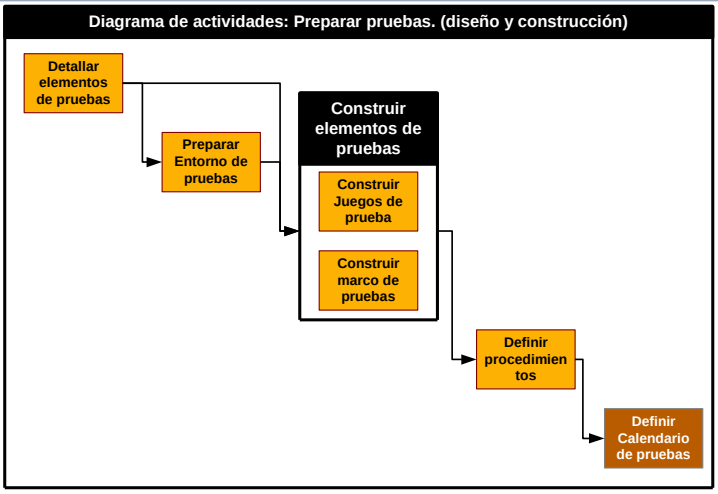
Entradas

- Requisitos del usuario
- Requisitos del sistema
- Casos de uso
- Historias de usuario
- Procesos de negocio
- Informe de análisis de riesgos

Objetivos

- Procesos de negocio
- Procesos operativos y de mantenimiento.
- Procedimientos de usuario
- Formularios
- Informes

Preparar las pruebas



Identificación y estructuración de los procedimientos de prueba

- Con los casos de prueba ya diseñados, pasas a identificar qué procedimientos se han de seguir para realizar los casos de prueba.
- En un documento especificas cómo realizar el proceso.
- Por economía, debes intentar reutilizar los procedimientos de prueba ya existentes, aunque sea necesario adaptarlos o modificarlos.
- De la misma forma, si creas un nuevo procedimiento intenta que sea reutilizable para varios casos de pruebas.
- Esto te permitirá usar un conjunto reducido de procedimientos de pruebas con rapidez y precisión para muchos casos de prueba.

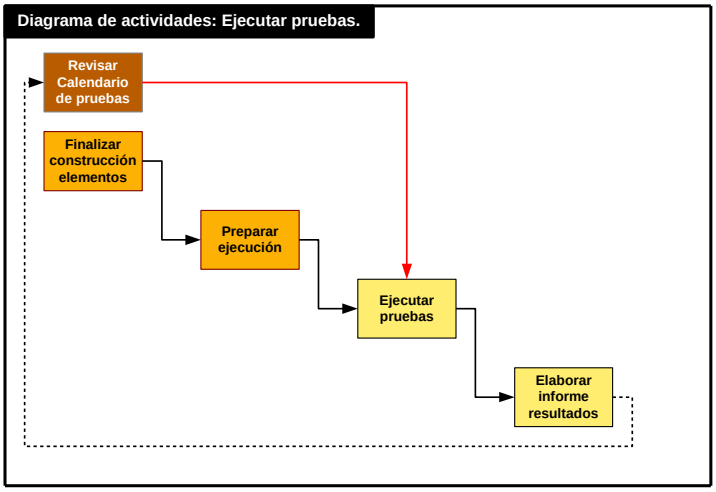
Implementar las pruebas

- El propósito de la implementación de las pruebas es automatizar los procedimientos de pruebas creando componentes de prueba (módulos conductores y módulos resguardo o auxiliares) y el repositorio o diccionario de datos de prueba.
- Esto lo harás siempre que el procedimiento se pueda automatizar (no siempre se puede, por ejemplo en el caso de las pruebas de interfaz de usuario) y no dispongas de herramientas específicas de pruebas o no puedas reutilizar componentes previamente desarrollados.
- La implementación de una prueba requiere un desarrollo completo.

Implementar las pruebas

- Partiendo del procedimiento de prueba como documento de requisitos debes realizar el análisis, diseño, implementación y prueba del componente de prueba.
- Aunque parezca surrealista, no es tan infrecuente como se podría pensar que el componente de prueba falle, invalidando la prueba para la que está diseñado.
- Los componentes de pruebas usan a menudo grandes cantidades de datos de entrada y generan, a su vez, grandes cantidades de datos de salida como resultado.
- Es por tanto deseable que incorpores como requisito el uso de bases de datos y hojas de cálculo donde se definan los elementos de entrada y salida del componente, que permitan la visualización e interpretación de los mismos de una forma clara e intuitiva.

Ejecutar las pruebas



Realizar las pruebas de integración

- Para realizar las pruebas de integración llevas a cabo los siguientes pasos:
  1. Realizas las pruebas siguiendo los pasos indicados el procedimiento de prueba.
  2. Comparas los resultados obtenidos con los esperados e investigas aquellos que no coinciden con lo esperado.
  3. Informas de los defectos a los responsables de los componentes que crees que contienen fallos.
  4. Recopilas los defectos para su posterior evaluación y uso en las pruebas de regresión.

Realizar la prueba de sistema

- Puedes comenzar con la prueba de sistema cuando las pruebas de integración indiquen que el sistema cumple los objetivos de calidad, fijados en el plan de prueba, para la integración (por ejemplo: el 98% de los casos de prueba se ejecutan con el resultado esperado).
- La prueba de sistema la realizas siguiendo los mismos pasos que en las pruebas de integración.

Evaluar la prueba

- Ya por último, pero no menos importante, realizas la evaluación de las pruebas.
- La evaluación la realizas estudiando los resultados y defectos obtenidos en las pruebas.
- Dicho estudio te permite preparar unas métricas que determinen el nivel de calidad y el espectro de pruebas.
- El elenco de métricas de calidad es muy amplio permitiendo medir la fiabilidad, seguridad y disponibilidad, entre otras, del sistema.

Evaluar la prueba

- La cobertura de pruebas indica el porcentaje de casos de prueba que han sido ejecutados y el porcentaje de código que ha sido probado.
- Basándote en el análisis de las tendencias de los defectos puedes sugerir:
  - Realizar pruebas adicionales.
  - Rebajar los criterios de calidad.
  - Aislar partes del sistema cuya calidad parece aceptable para su entrega, dejando el resto para ser revisado y probado de nuevo.
- El resultado de la evaluación lo debes recoger en un documento con todas tus conclusiones, métricas y sugerencias.

Características de los errores

- **Consecuencias del error**: no es lo mismo una cabecera errónea que un fallo en el suministro eléctrico.
- **Frecuencia del error**: ¿cuántas veces se produce un error? Evidentemente, será necesario prestar una mayor atención a los errores más comunes.
- **Coste de corrección**: este coste es una combinación del coste de descubrir el error y del coste de corregirlo. Además, este coste crece exponencialmente según se avanza en el proceso de desarrollo.
- **Coste de instalación**: especialmente en entornos distribuidos, a la hora tanto de instalar un sistema como de implantar cualquier corrección sobre el mismo puede llegar a ser un factor muy importante.
- **Origen causal**: fallo ← defecto ← causa.

Tipos de error

- Requerimientos
- Funcionales
- Estructurales
- Procedurales:
  - control,
  - lógica,
  - proceso,
  - inicialización
- Datos
- Implementación
- Interfaces
- Plataforma

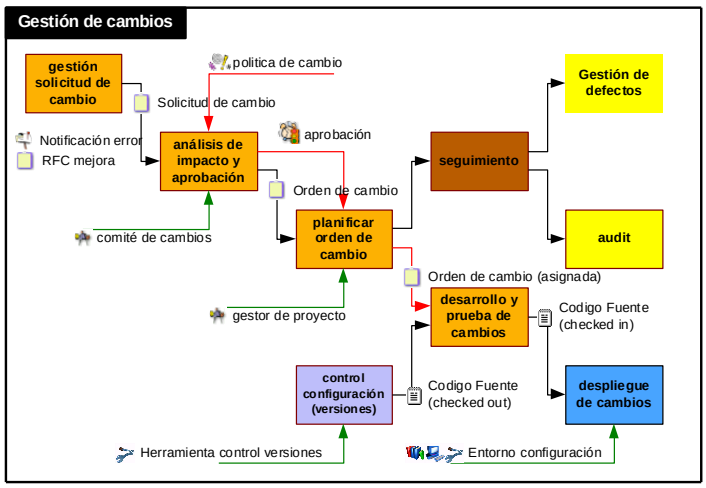
Gestión de Cambios

- En el último paso de la realización de las pruebas y en el documento de evaluación de las pruebas has informado de los defectos encontrados.
- La corrección de dichos defectos desencadena una solicitud de cambio.
- Dichos cambios deben ser gestionados.
- El control de cambios para el software es fundamental para el control de costes así como para la estabilidad de la tecnología.
- Para desarrollar software de calidad, los requerimientos deben estar completamente establecidos y mantenidos con una estabilidad razonable en el ciclo de desarrollo

Gestión de Cambios

- Los cambios de diseño y código deben ser realizados para corregir problemas encontrados en el desarrollo y pruebas, pero estos deben ser cuidadosamente introducidos.
- Si los cambios no son controlados, entonces el diseño ordenado, la implementación y las pruebas son imposibles y el plan de calidad no puede ser efectivo.
- El siguiente esquema muestra la secuencia de actividades para realizar la gestión de cambios:

Gestión de Cambios



PRUEBAS EN ANGULAR

Ingeniería de Software

- El JavaScript es un lenguaje muy poco apropiado para trabajar en un entorno de calidad de software.
- En descargo del lenguaje JavaScript y de su autor, Brendan Eich, hay que decir que los problemas que han forzado esta evolución del lenguaje (así como las críticas ancestrales de la comunidad de desarrolladores) vienen dados por lo que habitualmente se llama "morir de éxito".
- Jamás se pensó que un lenguaje que Eich tuvo que montar en 12 días como una especie de "demo" para Mozilla, pasase a ser omnipresente en miles de millones de páginas Web.
- O como el propio Hejlsberg comenta:

  "JavaScript se creó –como mucho- para escribir cien o doscientas líneas de código, y no los cientos de miles necesarias para algunas aplicaciones actuales."

Principios fundamentales

- Las pruebas exhaustivas no son viables
- El proceso de pruebas no puede demostrar la ausencia de defectos
- Las pruebas no garantizan ni mejoran la calidad del software
- Las pruebas tienen un coste
- Hay que ejecutar las pruebas bajo diferentes condiciones
- Inicio temprano de pruebas

Desarrollo Guiado por Pruebas (TDD)

- El Desarrollo Guiado por Pruebas, es una técnica de programación (definida por KentBeck); consistente en desarrollar primero el código que pruebe una característica o funcionalidad deseada antes que el código que implementa dicha funcionalidad.
- El objetivo a lograr es que no exista ninguna funcionalidad que no esté avalada por una prueba.
- Lo primero que hay que aprender de TDD son sus reglas básicas:
  - No añadir código sin escribir antes una prueba que falle
  - Eliminar el Código Duplicado empleando Refactorización

Ritmo TDD

- TDD invita a seguir una serie de tareas ordenadas, que a menudo se denomina ritmo TDD, y que se basa en los siguientes pasos:
  1. Escribir una prueba que demuestre la necesidad de escribir código.
  2. Escribir el mínimo código para que el código de pruebas compile
  3. Implementar exclusivamente la funcionalidad demandada por las pruebas
  4. Mejorar el código (Refactoring) sin añadir funcionalidad
  5. Volver al primer paso
- Este ritmo permite formalizar las tareas que se han de realizar para conseguir un código fácil de mantener, bien diseñado y que se puede probar automáticamente.

Las tres partes del test: AAA

- Arrange (Preparar):
  - Inicializa objetos y establece el valor de los datos que se pasa al método en pruebas de tal forma que los resultados sean predecibles.
- Act (Actuar)
  - Invoca al método a probar con los parámetros preparados.
- Assert (Afirmar)
  - Comprobar si la acción del método probado se comporta de la forma prevista. Puede tomar la forma de:
    - Aserción: Es una afirmación que se hace sobre el resultado y puede ser cierta o no.
    - Expectativa: Es la expresión del resultado esperado que puede cumplirse o no.

Arrange (Preparar)

- **Fixture**: Es el término se utiliza para hablar de los datos de contexto de las pruebas, aquellos que se necesitan para construir el escenario que requiere la prueba.
- **Dummy**: Objeto que se pasa como argumento pero nunca se usa realmente. Normalmente, los objetos dummy se usan sólo para rellenar listas de parámetros.
- **Fake**: Objeto que tiene una implementación que realmente funciona pero, por lo general, usa una simplificación que le hace inapropiado para producción (como una base de datos en memoria por ejemplo).
- **Stub**: Objeto que proporciona respuestas predefinidas a llamadas hechas durante los tests, frecuentemente, sin responder en absoluto a cualquier otra cosa fuera de aquello para lo que ha sido programado. Los stubs pueden también grabar información sobre las llamadas.
- **Mock**: Objeto preprogramado con expectativas que conforman la especificación de cómo se espera que se reciban las llamadas. Son más complejos que los stubs aunque sus diferencias son sutiles.

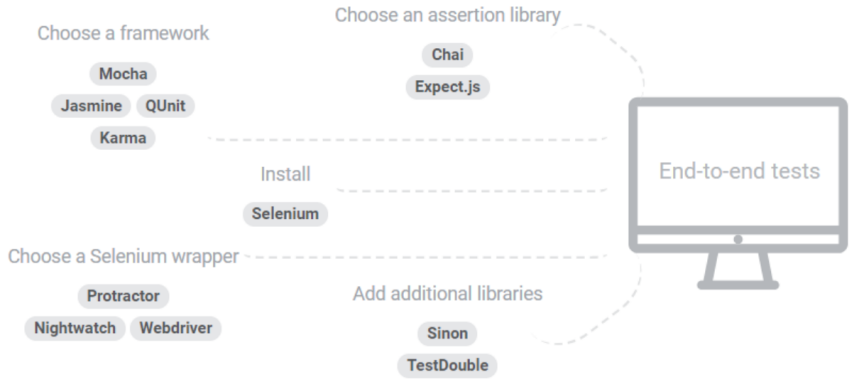
Beneficios de TDD

- Reducen el número de errores y bugs ya que éstos, aplicando TDD, se detectan antes incluso de crearlos.
- Facilitan entender el código y que, eligiendo una buena nomenclatura, sirven de documentación.
- Facilitan mantener el código:
  - Protege ante cambios, los errores que surgen al aplicar un cambio se detectan (y corrigen) antes de subir ese cambio.
  - Protegen ante errores de regresión (rollbacks a versiones anteriores).
  - Dan confianza.
- Facilitan desarrollar ciñéndose a los requisitos.
- Ayudan a encontrar inconsistencias en los requisitos
- Ayudan a especificar comportamientos
- Ayudan a refactorizar para mejorar la calidad del código (Clean code)
- A medio/largo plazo aumenta (y mucho) la productividad.

Herramientas y tecnologías

Tecnología	Propósito
Jasmine	El marco de trabajo Jasmine proporciona todo lo necesario para escribir pruebas unitarias. Cuenta con una página HTML que ejecuta pruebas en el navegador.
Utilidades Angular de pruebas	Las utilidades Angular de pruebas crean casos de prueba para el código de la aplicación Angular bajo prueba. Permiten añadir y controlar partes de la aplicación a medida que interactúan dentro del entorno Angular.
Karma	El lanzador de pruebas Karma es ideal para escribir y ejecutar pruebas unitarias, mientras se desarrolla la aplicación. Puede ser una parte integral de los procesos de desarrollo e integración continua del proyecto.
Protractor	Permite escribir y ejecutar pruebas de extremo a extremo (E2E), para explorar la aplicación tal y como los usuarios la experimentan. En las pruebas E2E: en un proceso se ejecuta la aplicación real y en un segundo proceso se ejecuta las pruebas Protractor, que simulan el comportamiento del usuario y comprueban que la aplicación responde en el navegador tal y como se esperaba.
Selenium WebDriver	El Selenium es un conjunto de herramientas para automatizar los navegadores web, un robot que simula la interacción del usuario con el navegador.

Herramientas y tecnologías



PRUEBAS UNITARIAS

JSLint, JSHint y TSLint

- Los analizadores de código son herramientas que realizan la lectura del código fuente y devuelve observaciones o puntos en los que tu código puede mejorarse desde la percepción de buenas prácticas de programación y código limpio.
- JSHint es un analizador online de código JavaScript (basado en el JSLint creado por Douglas Crockford) que nos permitirá mostrar puntos en los que tu código no cumpla unas determinadas reglas establecidas de “código limpio“.
- El funcionamiento de JSHint es el siguiente: toma nuestro código, lo escanea y, si encuentra un problema, devuelve un mensaje describiéndolo y mostrando su ubicación aproximada.
- Para descargar e instalar:
  - npm install -g jshint
- Existen “plug-in” para la mayoría de los entornos de desarrollo (http://jshint.com). Se puede automatizar con GRUNT o GULP.

Jasmine

- Jasmine es un framework de desarrollo dirigido por comportamiento (behavior-driven development, BDD) para probar código JavaScript.
  - No depende de ninguna otra librería JavaScript.
  - No requiere un DOM.
  - Tiene una sintaxis obvia y limpia para que se pueda escribir pruebas fácilmente.
- Prácticamente se ha convertido en el estándar de facto para el desarrollo con JavaScript.
- Para su instalación “standalone”, descargar y descomprimir:
  - https://github.com/jasmine/jasmine/releases
- Mediante npm:
  - npm install -g jasmine

SpecRunner.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Jasmine Spec Runner v2.5.0</title>
  <link rel="shortcut icon" type="image/png" href="lib/jasmine-2.5.0/jasmine_favicon.png">
  <link rel="stylesheet" href="lib/jasmine-2.5.0/jasmine.css">
  <script src="lib/jasmine-2.5.0/jasmine.js"></script>
  <script src="lib/jasmine-2.5.0/jasmine-html.js"></script>
  <script src="lib/jasmine-2.5.0/boot.js"></script>
  <script type="text/javascript" src="angular.js"></script>
  <script type="text/javascript" src="angular-mocks.js"></script>
  <!-- include source files here... -->
  <script src="src/Player.js"></script>
  <script src="src/Song.js"></script>
  <!-- include spec files here... -->
  <script src="spec/SpecHelper.js"></script>
  <script src="spec/PlayerSpec.js"></script>
</head>
<body></body>
</html>
```

Suites

- Una “suite” es un nombre que describe al género o sección qué se va a pasar por un conjunto de pruebas unitarias, además es una herramienta que es el núcleo que se necesita para poder tener un orden en el momento de crear las pruebas.
- Las “suites” se crean con la función **describe**, que es una función global y con la cual se inicia toda prueba unitaria, además consta con dos parámetros:

```
describe("Una suite es sólo una función", function() {
//…
});
```

- El primer parámetro es una cadena de caracteres donde se define el nombre descriptivo de la prueba unitaria.
- El segundo parámetro es una función con el código que ejecutará la prueba de código.
- Se pueden anidar “suites” para estructurar conjuntos complejos y facilitar la legibilidad y la búsqueda, basta con crear un describe dentro de otro.

Especificaciones

- Una especificación contiene una o más expectativas (algo que se espera) que ponen a prueba el estado del código. Una expectativa de Jasmine es una afirmación que debe ser verdadera pero puede ser falsa.
- Una especificación con todas las expectativas verdaderas es una especificación que pasa la prueba, pero con una o más falsas es una especificación que falla.
- Las especificaciones se definen dentro de una Suite llamando a la función global del Jasmine **it** que, al igual que describe, recibe una cadena y una función. La cadena es el título de la especificación y la función es la especificación o prueba.
  - it("y así es una especificación", function() {
  - //…
  - });
- **describe** y **it** son funciones: pueden contener cualquier código ejecutable necesario para implementar la prueba y las reglas de alcance de JavaScript se aplican, por lo que las variables declaradas en un describe están disponibles para cualquier bloque it dentro de la suite.

Expectativas

- Las expectativas se construyen con la función expect que obtiene un valor real de una expresión y lo comparan mediante una función Matcher con un el valor esperado (constante).

  expect(valor actual).matchers(valor esperado);
- Los matchers son funciones que implementan comparaciones booleanas entre el valor actual y el esperado, ellos son los responsables de informar a Jasmine si la expectativa se cumple o es falsa.
- Cualquier matcher puede evaluarse como una afirmación negativa mediante el encadenamiento a la llamada expect de un not antes de llamar al matcher.

  expect(valor actual)**.not**.matchers(valor esperado);
- También existe la posibilidad de escribir matchers personalizados para cuando el dominio de un proyecto consiste en afirmaciones específicas que no se incluyen en los ya definidos.

Matchers

- .toEqual(y); verifica si ambos valores son iguales ==.
- .toBe(y); verifica si ambos objetos son idénticos ===.
- .toMatch(pattern); verifica si el valor pertenece al patrón establecido.
- .toBeDefined(); verifica si el valor está definido.
- .toBeUndefined(); verifica si el valor es indefinido.
- .toBeNull(); verifica si el valor es nulo.
- .toBeNaN(); verifica si el valor es NaN.
- .toBeCloseTo (n, d); verifica la precisión matemática (número de decimales).
- .toContain(y); verifica si el valor actual contiene el esperado.

Matchers

- .toBeTruthy(); verifica si el valor es verdadero.
- .toBeFalsy(); verifica si el valor es falso.
- .toBeLessThan(y); verifica si el valor actual es menor que el esperado.
- .toBeLessThanOrEqual (y); verifica si el valor actual es menor o igual que el esperado.
- .toBeGreaterThan(y); verifica si el valor actual es mayor que el esperado.
- .toBeGreaterThanOrEqual (y); verifica si el valor actual es mayor o igual que el esperado.
- .toThrow(); verifica si una función lanza una excepción.
- .toThrowError(e); verifica si una función lanza una excepción específica.

Forzar fallos

- La función fail(msg) hace que una especificación falle. Puede llevar un mensaje de fallo o error de un objeto como un parámetro.

```
describe("Una especificación utilizando la función a prueba", function() {
  var foo = function(x, callBack) {
    if (x) {
      callBack();
    }
  };
  it("no debe llamar a la devolución de llamada", function() {
    foo(false, function() {
      fail("Devolución de llamada ha sido llamada");
    });
  });
});
```

Montaje y desmontaje

- Para montar el escenario de pruebas suele ser necesario definir e inicializar un conjunto de variables. Para evitar la duplicidad de código y mantener las variables inicializadas en un solo lugar además de mantener la modularidad, Jasmine suministra las funciones globales :
  - beforeAll(fn) se ejecuta solo una vez antes empezar a ejecutar las especificaciones del "describe".
  - beforeEach(fn) se ejecuta antes de cada especificación dentro del "describe".
  - afterEach(fn) se ejecuta después de cada especificación dentro del "describe".
  - afterAll(fn) se ejecuta solo una vez después de ejecutar todas las especificaciones del "describe".

```
describe("operaciones aritméticas", function(){
    var cal;
    beforeEach(function(){ calc = new Calculadora(); });
    it("adicion", function(){ expect(calc.suma(4)).toEqual(4); });
    it("multiplicacion", function(){ expect(calc.multiplica(7)).toEqual(0); });
    // …
```

- Otra manera de compartir las variables entre una beforeEach, it y afterEach es a través de la palabra clave this. Cada expectativa beforeEach/it/afterEach tiene el mismo objeto vacío this que se restablece de nuevo a vacío para de la siguiente expectativa beforeEach/it/afterEach.

Desactivación parcial

- Las Suites se pueden desactivar renombrando la función describe por xdescribe. Estas suites y las especificaciones dentro de ellas se omiten cuando se ejecuta y por lo tanto sus resultados no aparecerán entre los resultados de la prueba.
- De igual forma, las especificaciones se desactivan renombrando it por xit, pero en este caso aparecen en los resultados como pendientes (pending).
- Cualquier especificación declarada sin un cuerpo función también estará marcada pendiente en los resultados.

```
it('puede ser declarada con "it", pero sin una función');
```

- Y si se llama a la función de pending en cualquier parte del cuerpo de las especificaciones, independientemente de las expectativas, la especificación quedará marcada como pendiente. La cadena que se pasa a pending será tratada como una razón y aparece cuando termine la suite.

```
it('se puede llamar a "pending"en el cuerpo de las especificaciones', function() {
    expect(true).toBe(false);
    pending('esto es por lo que está pendiente');
});
```

Ejecución de pruebas específicas

- En determinados casos (desarrollo) interesa limitar las pruebas que se ejecutan. Si se pone el foco en determinadas suites o especificaciones solo se ejecutaran las pruebas que tengan el foco, marcando el resto como pendientes.
- Las Suites se enfocan renombrando la función describe por fdescribe. Estas suites y las especificaciones dentro de ellas son las que se ejecutan.
- De igual forma, las especificaciones se enfocan renombrando it por fit.
- Si se enfoca una suite que no tiene enfocada ninguna especificación se ejecutan todas sus especificaciones, pero si tiene alguna enfocada solo se ejecutaran las que tengan el foco.
- Si se enfoca una especificación se ejecutara independientemente de que su suite esté o no enfocada.
- La funciones de montaje y desmontaje se ejecutaran si la suite tiene alguna especificación con foco.
- Si ninguna suite o especificación tiene el foco se ejecutaran todas las pruebas normalmente.

Espías

- Jasmine tiene funciones dobles de prueba llamados espías.
- Un espía puede interceptar cualquier función y hacer un seguimiento a las llamadas y todos los argumentos.

```
beforeEach(function() {
  fnc = spyOn(calc, 'suma');
  prop = spyOnProperty(calc, 'pantalla', 'set')
});
```

- Un espía sólo existe en el bloque describe o it en que se define, y se eliminará después de cada especificación.
- Hay comparadores (matchers) especiales para interactuar con los espías.
  - .toHaveBeenCalled() pasará si el espía fue llamado.
  - .toHaveBeenCalledTimes(n) pasará si el espía se llama el número de veces especificado.
  - .toHaveBeenCalledWith(…) pasará si la lista de argumentos coincide con alguna de las llamadas grabadas a la espía.
  - .toHaveBeenCalledBefore(esperado): pasará si el espía se llama antes que el espía pasado por parámetro.

Seguimiento de llamadas

- El proxi del espía añade la propiedad calls que permite:
  - all(): Obtener la matriz de llamadas sin procesar para este espía.
  - allArgs(): Obtener todos los argumentos para cada invocación de este espía en el orden en que fueron recibidos.
  - any(): Comprobar si se ha invocado este espía.
  - argsFor(índice): Obtener los argumentos que se pasaron a una invocación específica de este espía.
  - count(): Obtener el número de invocaciones de este espía.
  - first(): Obtener la primera invocación de este espía.
  - mostRecent(): Obtener la invocación más reciente de este espía.
  - reset(): Restablecer el espía como si nunca se hubiera llamado.
  - saveArgumentsByValue(): Establecer que se haga un clon superficial de argumentos pasados a cada invocación.

```
spyOn(foo, 'setBar');
expect(foo.setBar.calls.any()).toEqual(false);
foo.setBar();
expect(foo.setBar.calls.count()).toBy(1);
```


Cambiar comportamiento

- Adicionalmente el proxi del espía puede añadir los siguientes comportamiento:
  - callFake(fn): Llamar a una implementación falsa cuando se invoca.
  - callThrough(): Llamar a la implementación real cuando se invoca.
  - exec(): Ejecutar la estrategia de espionaje actual.
  - identity(): Devolver la información de identificación para el espía.
  - returnValue(valor): Devolver un valor cuando se invoca.
  - returnValues(... values): Devolver uno de los valores especificados (secuencialmente) cada vez que se invoca el espía.
  - stub(): No haga nada cuando se invoca. Este es el valor predeterminado.
  - throwError(algo): Lanzar un error cuando se invoca.

```
spyOn(foo, "getBar").and.returnValue(745);
spyOn(foo, "getBar").and.callFake(function(arguments, can, be, received) {
  return 745;
});
spyOn(foo, "forbidden").and.throwError("quux");
```

Karma

- Karma es una herramienta de línea de comandos JavaScript que se puede utilizar para generar un servidor web que carga el código fuente de la aplicación y ejecuta sus pruebas.
- Puede configurar Karma para funcionar contra una serie de navegadores, que es útil para estar seguro de que la aplicación funciona en todos los navegadores que necesita soportar.
- Karma se ejecuta en la línea de comandos y mostrará los resultados de sus pruebas en la línea de comandos una vez que se ejecute en el navegador.
- Karma es una aplicación NodeJS, y debe ser instalado a través de npm:
  - npm install karma karma-jasmine jasmine-core karma-chrome-launcher --save-dev
  - npm install -g karma-cli

Cobertura de código

- Con Angular CLI podemos ejecutar pruebas unitarias y crear informes de cobertura de código. Los informes de cobertura de código nos permiten ver que parte del código ha sido o no probada adecuadamente por nuestras pruebas unitarias.
- Para generar el informe de cobertura:

  ng test --watch=false --code-coverage
- Una vez que las pruebas se completen, aparecerá una nueva carpeta /coverage en el proyecto. Si se abre el archivo index.html en el navegador se debería ver un informe con el código fuente y los valores de cobertura del código.
- Usando los porcentajes de cobertura del código, podemos establecer la cantidad de código (instrucciones, líneas, caminos, funciones) que debe ser probado. Depende de cada organización determinar la cantidad de código que deben cubrir las pruebas unitarias.

Cobertura de código

- Para establecer un mínimo de 80% de cobertura de código cuando las pruebas unitarias se ejecuten en el proyecto, se debe configurar en karma.conf.js

```
coverageIstanbulReporter: {
  reports: [ 'html', 'lcovonly' ],
  fixWebpackSourcePaths: true,
  thresholds: { statements: 80, lines: 80, branches: 80, functions: 80  }
}
```

- Para generar siempre el informe de cobertura de código (sin usar --code-coverage), se debe configurar en angular.json

```
"test": {
   "options": {
    "codeCoverage": true
   }
  }
```


Depuración de pruebas

1. Seleccionar la ventana del navegador Karma.
2. Hacer clic en el botón DEBUG; se abre una nueva pestaña del navegador que permite volver a ejecutar las pruebas.
3. Abrir “Herramientas de Desarrollo” del navegador.
4. Seleccionar la sección de código fuentes.
5. Abrir el archivo con el código de prueba.
6. Establecer un punto de interrupción en la prueba.
7. Actualizar el navegador, que se detiene en el punto de interrupción.

Pruebas Angular

- Angular, por defecto, se encuentra alineado con la calidad de software.
- Cuando Angular-CLI crea un nuevo proyecto:
  - Descarga TSLint, Jazmine, Karma y Protractor (e2e)
  - Configura el entorno de pruebas
  - Habilita un servidor Karma de pruebas continuas (puerto: 9876)
- Así mismo, cuando genera un nuevo elemento, crea el correspondiente fichero de pruebas, usando .spec como sub extensión.
- Para ejecutar el servidor de pruebas:
  - **ng test** *--code-coverage (alias: -cc)*
  - ng test --single-run (alias: -sr)
- No se debe cerrar la instancia de Chrome mientras duren las pruebas.
- Angular suministra una serie de clases, funciones, mock y módulos específicos para las pruebas, comúnmente denominadas Utilidades Angular para pruebas.
- Permite la creación tanto de pruebas unitarias aisladas como casos de prueba que interactúan dentro del entorno Angular.

Pruebas Unitarias Aisladas

- Las Pruebas Unitarias Aisladas examinan una instancia de una clase por sí misma sin ninguna dependencia Angular o de los valores inyectados.
- Se crea una instancia de prueba de la clase con new, se le suministran los parámetros al constructor y se prueba la superficie de la instancia.
- Se pueden escribir pruebas unitarias aisladas para pipes y servicios.
- Aunque también se puede probar los componentes de forma aislada, las pruebas aisladas no revelan cómo interactúan entre si los elementos Angular. En particular, no pueden revelar cómo una clase de componente interactúa con su propia plantilla o con otros componentes.

Pruebas Unitarias Aisladas

- De servicios sin dependencias
  - let srv = new MyService();
- De servicios con dependencias
  - let srv = new MyService(new OtherService());
- De Pipes
  - let pipe = new MyPipe();
  - expect(pipe.transform('abc')).toBe('Abc');
- De la clase del componente:
  - let comp = new MyComponent();
- De la clase del componente con dependencias :
  - let comp = new MyComponent(new MyService());

Utilidades Angular para pruebas

- Para realizar pruebas dentro del contexto de Angular, las Utilidades Angular para pruebas cuentan con las clases TestBed, ComponentFixture, DebugElement y By, así como varias funciones de ayuda para sincronizar, inyectar, temporizar, …
- Para probar los componentes, lo mas correcto es crear dos juegos de pruebas, a menudo en el mismo archivo de especificaciones.
  - Un primer conjunto de pruebas aisladas que examinan la corrección de la clase del componente.
  - Un segundo conjunto de pruebas que examina como se comporta el componente dentro del Angular, como interactúa con las plantillas, si actualiza el DOM y colabora con el resto de la aplicación.

TestBed

- TestBed representa un módulo Angular para la prueba, proporciona el medio ambiente del módulo para la clase que desea probar. Extrae el componente a probar desde su propio módulo de aplicación y lo vuelve a conectar al módulo de prueba construido a medida, de forma dinámica, específicamente para una serie de pruebas.
- El método configureTestingModule reemplaza a la anotación @NgModule en la declaración del módulo. Recibe un objeto @NgModule que puede tener la mayoría de las propiedades de metadatos de un módulo normal de Angular.
- El estado base incluye la configuración del módulo de prueba predeterminado con las declaraciones (componentes, directivas y pipes) y los proveedores (servicios inyectables) necesarios para el entorno de prueba.
- El método configureTestingModule se suele invocar dentro de un método beforeEach de modo que TestBed pueda restablecer el estado base antes de cada ejecuciones de pruebas.

Preparación de la prueba

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { DebugElement } from '@angular/core';
import { By } from '@angular/platform-browser';

import { MyComponent } from './my.component';

describe('Prueba de MyComponent', () => {
  let fixture: ComponentFixture<MyComponent>;
  let comp:    MyComponent;
  let de:      DebugElement;
  let tag:     HTMLElement;

  beforeEach(() => {
    TestBed.configureTestingModule({ declarations: [ MyComponent ],  });
    // …
   }
```

Pruebas poco profundas

- En TestBed.configureTestingModule hay que declarar todas las dependencias del componente a probar: otros componentes, pipes y directivas propias, proveedores de los servicios utilizados, incluso los módulos importados.

```
TestBed.configureTestingModule({
  imports: [MyCoreModule],
  declarations: [MyComponent, OtherComponent, MyPipe, MyDirective, ...],
})
```

- Agregando NO_ERRORS_SCHEMA (en @angular/core) a los metadatos del esquema del módulo de prueba se indica al compilador que ignore los elementos y atributos no reconocidos. Ya no es necesario declarar los elementos de plantilla irrelevantes.

```
TestBed.configureTestingModule({
  declarations: [ MyComponent ],
  schemas:      [ NO_ERRORS_SCHEMA ]
});
```


ComponentFixture y DebugElement

- Un ComponentFixture es un contexto (fixture) que envuelve el componente creado en el entorno de prueba.

```
fixture = TestBed.createComponent(MyComponent);
```

- El fixture proporciona acceso a si mismo, a la instancia del componente y al envoltorio del elemento DOM del componente.

```
comp = fixture.componentInstance; // instancia del componente
```

- El DebugElement es un envoltorio del elemento DOM del componente o de los elementos localizados.

```
de = fixture.debugElement;
tag = de.nativeElement;
```

***NOTA**: Una vez ejecutado el método createComponent se cierra la configuración del TestBed, si se intenta cambiar la configuración dará un error.*

Consultar DebugElement

- La clase By es una utilidad Angular de pruebas para consultar el árbol del DOM. El método estático By.css utiliza un selector CSS estándar para generar un predicado (función que devuelve un valor booleano) que filtra de la misma manera que un selector de jQuery.
- Un predicado de consulta recibe un DebugElement y devuelve true si el elemento cumple con los criterios de selección.
- La clase DebugElement dispone de métodos para, utilizando una función de predicado, buscar en todo el árbol DOM del fixture:
  - El método query devuelve el primer elemento que satisface el predicado.
  - El método queryAll devuelve una matriz de todos los DebugElement que satisfacen el predicado.
- La propiedad nativeElement de DebugElement obtiene el elemento DOM.

```
let tag: HTMLElement = fixture.debugElement.query(By.css('#myId')).
    nativeElement;
```


Consultas By

- Recuperar un componente con elemento HTML:

```
tag  = fixture.debugElement.query(By.css('my-component'));
```

- Recuperar el valor de la primera etiqueta:

```
it('should render title in a h1 tag', async(() => {
  fixture.detectChanges();
  const tag = fixture.debugElement.query(By.css('h1'));
  expect(tag.nativeElement.textContent).toContain('Welcome to app!!');
 }));
```

- Recuperar los elementos de un listado:

```
it('renders the list on the screen', () => {
  fixture.detectChanges();
  const li = fixture.debugElement.queryAll(By.css('li'));
  expect(li.length).toBe(2);
});
```

triggerEventHandler

- El método DebugElement.triggerEventHandler permite simular que el elemento ha lanzado un determinado evento.

```
fixture.debugElement.triggerEventHandler('click', eventData);
```

- Según sea el evento, eventData representa el objeto event del DOM o el valor emitido. En algunos casos el eventData es obligatorio y con una determinada estructura.
- Dado que los eventos están vinculados a comandos, suele ser preferible invocar directamente a los métodos comando de la clase componente.
- Puede ser necesario para probar la interacción de determinadas directivas con el elemento y para probar las vinculaciones @HostListener.

Detección de cambios

- La prueba puede decir a Angular cuándo realizar la detección de cambios, lo que provoca el enlace de datos y la propagación de las propiedades al elemento DOM.

```
fixture.detectChanges()
```

- Cuando se desea que los cambios se propaguen automáticamente sin necesidad de invocar fixture.detectChanges():

```
import { ComponentFixtureAutoDetect } from '@angular/core/testing';
// …
TestBed.configureTestingModule({
  declarations: [ MyComponent ],
  providers: [
    { provide: ComponentFixtureAutoDetect, useValue: true }
  ]
})
```

- El servicio ComponentFixtureAutoDetect responde a las actividades asíncronas como la resolución de la promesa, temporizadores y eventos DOM. Pero una actualización directa, síncrona de una propiedad de componente es invisible. La prueba debe llamar fixture.detectChanges() manualmente para desencadenar otro ciclo de detección de cambios.

Inyección de dependencias

```
TestBed.configureTestingModule({
    declarations: [ MyComponent ],
    providers:    [ MyService ]
  });
```

- Un componente bajo prueba no tiene por que ser inyectado con servicios reales, por lo general es mejor si son dobles de pruebas (stubs, fakes, spies o mocks), dado que el propósito de la especificación es probar el componente y no el servicio o servicios reales que pueden ser el origen del error.

```
providers:    [ {provide: MyService, useValue: MyServiceFake },
        { provide: Router, useClass: RouterStub} ]
```

- Si la pruebas necesita tener acceso al servicio inyectado, la forma más segura es obtener el servicio desde el fixture.

```
srv = fixture.debugElement.injector.get(MyService);
```

- También se puede obtener el servicio desde TestBed:

```
srv = TestBed.get(MyService);
```


inject

- La función inject es una de las utilidades Angular de prueba.
- Inyecta servicios en la función especificación donde se los puede alterar, espiar y manipular.
- La función inject tiene dos parámetros:
  - Una matriz de tokens de inyección de dependencias Angular.
  - Una función de prueba cuyos parámetros corresponden exactamente a cada elemento de la matriz de tokens de inyección.

```
it('demo inject', inject([Router], (router: Router) => {
  // ...
}));
```

- La función inject utiliza el inyector del módulo TestBed actual y sólo puede devolver los servicios proporcionados a ese nivel. No devuelve los servicios de los proveedores de componentes.

***NOTA**: Una vez ejecutado el método createComponent se cierra la configuración del TestBed, si se intenta cambiar la configuración dará un error.*

Sobre escritura de @Component

- En algunos casos, sobre todo en la inyección de dependencias a nivel de componentes es necesario sobreescibir la definición del componente.
- La estructura MetadataOverride establece las propiedades de @Component a añadir, modificar o borrar:

  type MetadataOverride = { add?: T; remove?: T; set?: T; };
- Donde T son las propiedades de la anotación:

  selector?: string;

  template?: string; ó templateUrl?: string;

  providers?: any[];

  …

Sobre escritura de @Component

- El método overrideComponent recibe el componente a modificar y los metadatos con las modificaciones:

```
TestBed.configureTestingModule({
      declarations: [ MyComponent, MyChildComponent],
      providers:    [ { provide: Router, useClass: RouterStub} ]
   })
   .overrideComponent(MyChildComponent, {
     set: {
       providers: [
         { provide: MyService, useClass: MyServiceSpy }
       ]
     }
   });
```

Creación asíncrona

- Si el componente tiene archivos externos de plantillas y CSS, que se especifican en las propiedades templateUrl y styleUrls, supone un problema para las pruebas.
- El método TestBed.createComponent es síncrono.
- Sin embargo, el compilador de plantillas Angular debe leer los archivos externos desde el sistema de archivos antes de que pueda crear una instancia de componente. Eso es una actividad asincrónica.
- El método compileComponents devuelve una promesa para que se puedan realizar tareas adicionales inmediatamente después de que termine.
- La función async es una de las utilidades Angular de prueba que esconde la mecánica de ejecución asíncrona, envuelve una función de especificación en una zona de prueba asincrónica, la prueba se completará automáticamente cuando se finalicen todas las llamadas asíncronas dentro de esta zona.

Preparación de la prueba asíncrona

```
import { TestBed, async } from '@angular/core/testing';
// …
describe('AppComponent', () => {
  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ AppComponent  ],
    }).compileComponents();
  }));

  it('should create the app', async(() => {
    const fixture = TestBed.createComponent(AppComponent);
    const app = fixture.debugElement.componentInstance;
    expect(app).toBeTruthy();
  }));
```

Inyección de servicios asíncronos

- Muchos servicios obtienen los valores de forma asíncrona. La mayoría de los servicios de datos hacen una petición HTTP a un servidor remoto y la respuesta es necesariamente asíncrona.
- Salvo cuando se estén probando los servicios asíncronos, las pruebas no deben hacer llamadas a servidores remotos. Las pruebas deberían emular este tipo de llamadas.
- Disponemos de las siguientes técnicas:
  - Sustituir el servicio asíncrono por un servicio síncrono.
  - Sustituir el método asíncrono por un espía.
  - Crear una zona de pruebas asíncrona.
  - Crear una falsa zona de pruebas asíncrona.

Espías

- Los espías de Jasmine permiten interceptar y sustituir métodos de los objetos.
- Mediante el espía se sustituye el método asíncrono de tal manera que cualquier llamada al mismo recibe una promesa resuelta de inmediato con un valor de prueba (stub).
- El espía no invoca el método real, por lo que no entra en contacto con el servidor.
- En lugar de crear un objeto de servicio sustituto, se inyecta el verdadero servicio y se sustituye el método crítico con un espía Jasmine.

Espías

```
beforeEach(() => {
  TestBed.configureTestingModule({
    declarations: [ MyComponent ],
    providers:    [ MyService ],
  });
  fixture = TestBed.createComponent(MyComponent);
  comp    = fixture.componentInstance;
  srv = fixture.debugElement.injector.get(MyService);
  spy = spyOn(srv, 'myAsyncMethod')
      .and.returnValue(Promise.resolve('result value'));
  // ...
});
it('Prueba asíncrona con espia', () => {
  // ...
  fixture.detectChanges();
  expect(...).matcher(...);
  expect(spy.calls.any()).toBe(true, 'myAsyncMethod called');
  expect(spy.calls.count()).toBe(1, 'stubbed method was called once');
  expect(myService.myAsyncMethod).toHaveBeenCalled();
});
```


whenStable

- En algunos casos, la prueba debe esperar a que la promesa se resuelva en el siguiente ciclo de ejecución del motor de JavaScript.
- En este escenario la prueba no se tiene acceso directo a la promesa devuelta por la llamada del método asíncrono dado que está encapsulada en el interior del componente, inaccesibles desde la superficie expuesta.
- Afortunadamente, la función async genera una zona de prueba asíncrona, que intercepta todas las promesas emitidas dentro de la llamada al método asíncrono sin importar dónde se producen.
- El método ComponentFixture.whenStable devuelve su propia promesa que se resuelve cuando todas las actividades asíncronas pendientes dentro de la prueba se han completado (cuando sea estable).

```
it('Prueba asíncrona cuando sea estable', async(() => {
  fixture.detectChanges();
  fixture.whenStable().then(() => {
    fixture.detectChanges();
    expect(...).matcher(...);
  });
}));
```

fakeAsync

- La función fakeAsync, otra de las utilidades Angular de prueba, es una alternativa a la función async. La función fakeAsync permite un estilo de codificación secuencial mediante la ejecución del cuerpo de prueba en una zona especial de ensayo propia de fakeAsync, haciendo que la prueba aparezca como si fuera síncrona.
- Se apoya en la función tick (), que simula el paso del tiempo hasta que todas las actividades asincrónas pendientes concluyen, similar al wait en concurrencia. Sólo se puede invocar dentro del cuerpo de fakeAsync, no devuelve nada, no hay promesa que esperar.

```
it('Prueba asíncrona cuando con fakeAsync', fakeAsync(() => {
  fixture.detectChanges();
  tick();
  fixture.detectChanges();
  expect(...).matcher(...);
 }));
```


Componentes contenidos

- Para probar los componentes simulando que están contenidos en una plantilla es necesario crear un componente Angular para la prueba (wrapper):

```
@Component({
  template: `<my-component [myInput]="MyInput"
      (myOutput)="onMyOutput($event)"></my-component>`
})
class TestHostComponent {
  @ViewChild(MyComponent) myComponent: MyComponent;
  MyInput: any = null;
  MyOutput: any;
  onMyOutput(rslt: any) { this.MyOutput = rslt; }
}
```

Componentes contenidos

- Para posteriormente instanciarlo:

```
beforeEach( async(() => {
  TestBed.configureTestingModule({
    declarations: [ MyComponent, TestHostComponent ],
  }).compileComponents();
}));

beforeEach(() => {
  // create TestHostComponent instead of MyComponent
  fixture  = TestBed.createComponent(TestHostComponent);
  testHost = fixture.componentInstance;
  tag  = fixture.debugElement.query(By.css('my-component'));
  fixture.detectChanges(); // trigger initial data binding
});
```


Entradas y Salidas

- Entrada: Se modifican las entradas a través del componente de pruebas y se comprueba que las modificaciones se reflejan en el componente contenido.

```
it('input test', () => {
  testHost.MyInput = '666';
  fixture.detectChanges();
  expect(testHost.myComponent.getInit()).toBe('666');
});
```

- Salida: Se interactúa con el componente contenido para que se disparen los eventos de salida y se comprueba en el componente de pruebas que las modificaciones se han reflejado en el.

```
it('output test', () => {
  testHost.myComponent.exec();
  fixture.detectChanges();
  expect(testHost.MyOutput).toBe('666');
});
```

Pruebas de observables

- Para poder probar los observables es necesario:
  - Crear una zona asíncrona
  - Convertir el observable en una promesa
  - Definir las expectativas dentro del .then de la promesa.

```
import 'rxjs/add/operator/toPromise';

it('get http', async(inject([HttpClient], (http: HttpClient) => {
  http.get(url)
    .toPromise().then(
      data => { expect(data).toBeTruthy(); },
      err => { fail(); }
    );
})));
```


Dobles de prueba

- La regla de oro de las pruebas unitarias, es que una unidad (unit) tiene que ser testeada sin utilizar ninguna de sus dependencias.
- Eso quiere decir que si una unidad tiene dependencias hay que reemplazarlas por mocks.
- Un ejemplo de mock de un servicio sería:

```
class MyServiceSpy {
  getData = jasmine.createSpy('getData').and.callFake(() => {
    return of([{ id: 0, name: 'Uno'}, { id: 1, name: 'Dos' }]);
  });
}
```

- Para crear las especificación:

```
it('fetches all data', () => {
  // …
  expect(instance.names.length).toBe(2);
  expect(MyService.getData).toHaveBeenCalled();
});
```

Doble de prueba Observable

```
import { of, Observable } from 'rxjs';

export class DAOServiceMock {
  constructor(private listado: Array<any>) { }
  query(): Observable<any> { return of(this.listado); }
  get(id: number) { return of(this.listado[0]); }
  add(item: any) { return of(item); }
  change(id: number, item: any) { return of(item); }
  remove(id: number) { return of(id); }
}

{provide: MyDAOService, useValue: new DAOServiceMock([{ id: 0,
      name: 'Uno'}, { id: 1, name: 'Dos' }])}
```


Prueba de peticiones HTTP

- La biblioteca de pruebas HTTP de Angular está diseñada siguiendo un patrón de pruebas donde la especificación empieza haciendo las solicitudes.
- Después de eso, las pruebas esperan a que ciertas solicitudes hayan sido o no realizadas, se cumplan determinadas afirmaciones contra esas solicitudes y, finalmente, se proporcionan respuestas "descargando" cada solicitud esperada, lo que puede activar más solicitudes nuevas, etc.
- Al terminar, se pueden verificar que la aplicación no ha hecho peticiones inesperadas.
- Para disponer de la biblioteca de pruebas HTTP:

  imports: [HttpClientTestingModule,],

Prueba de peticiones HTTP

- El módulo instala un mock que sustituye el acceso real al servidor.
- HttpTestingController es el controlador que se inyecta en las pruebas, permite la inspección y el volcado de las solicitudes.

```
it('query', inject([DAOService, HttpTestingController], (dao: DAOService,
httpMock: HttpTestingController) => {
  dao.query().subscribe(
    data => { expect(data.length).toEqual(2); },
    data => { fail(); }
  );
  const req = httpMock.expectOne('http://localhost:4321/data');
  expect(req.request.method).toEqual('GET');
  req.flush([{ name: 'Data' }, { name: 'Data2' }]);
  httpMock.verify();
 }));
```


Enrutado

- Para disponer del enrutador se utilizará el módulo:

```
imports: [ // …
    RouterTestingModule.withRoutes([
      {path: '', component: HomeCmp},
      {path: 'simple', component: SimpleCmp}
    ])]
```

- Para crear y registrar un sustituto del Router:

```
class RouterStub {
  navigateByUrl(url: string) { return url; }
  navigate(commands: Array<any>) { return url; }
}
{ provide: Router, useClass: RouterStub },
```

- Para interceptar las llamadas al Router:

```
it('Demo ROUTER', inject([Router], (router: Router) => { // ...
  const spy = spyOn(router, 'navigateByUrl');
  // ...
  const navArgs = spy.calls.first().args[0];
  expect(navArgs).toBe(...);
}));
```

http://www.protractortest.org/

PRUEBAS E2E

Introducción

- A medida que las aplicaciones crecen en tamaño y complejidad, se vuelve imposible depender de pruebas manuales para verificar la corrección de las nuevas características, errores de captura y avisos de regresión.
- Las pruebas unitarias son la primera línea de defensa para la captura de errores, pero a veces las circunstancias requieran la integración entre componentes que no se pueden capturar en una prueba unitarias.
- Las pruebas de extremo a extremo (E2E: end to end) se hacen para encontrar estos problemas.
- El equipo de Angular ha desarrollado Protractor que simula las interacciones del usuario con el interfaz (navegador) y que ayudará a verificar el estado de una aplicación Angular.
- Protractor es una aplicación Node.js para ejecutar pruebas de extremo a extremo que también están escritas en JavaScript y se ejecutan con el propio Node.
- Protractor utiliza WebDriver para controlar los navegadores y simular las acciones del usuario.

Introducción

- Protractor utiliza Jasmine para su sintaxis prueba.
- Al igual que en las pruebas unitarias, el archivo de pruebas se compone de uno o más bloques describe de it que describen los requisitos de su aplicación.
- Las bloques it están hechos de comandos y expectativas.
- Los comandos indican a Protractor que haga algo con la aplicación, como navegar a una página o hacer clic en un botón.
- Las expectativas indican a Protractor afirmaciones sobre algo acerca del estado de la aplicación, tales como el valor de un campo o la URL actual.
- Si alguna expectativa dentro de un bloque it falla, el ejecutor marca en it como "fallido" y continúa con el siguiente bloque.
- Los archivos de prueba también pueden tener bloques beforeEach y afterEach, que se ejecutarán antes o después de cada bloque it, independientemente de si el bloque pasa o falla.

Arquitectura

- Protractor es un marco de prueba de extremo a extremo para aplicaciones AngularJS y Angular. Protractor es un programa Node.js que admite los marcos de prueba de Jasmine y Mocha.
- Selenium es un marco de automatización de navegadores. Selenium incluye el servidor Selenium, las API de WebDriver y los controladores del navegador WebDriver.
- Protractor funciona junto con Selenium para proporcionar una infraestructura de prueba automatizada que puede simular la interacción de un usuario con una aplicación Angular que se ejecuta en un navegador o dispositivo móvil.
- Es importante tener en cuenta lo siguiente:
  - Protractor es una envoltura alrededor de WebDriverJS, los enlaces de JavaScript para la API de Selenium WebDriver.
  - Los comandos de WebDriver son asíncronos. Se programan en un flujo de control y devuelven promesas, no valores primitivos.
  - Los scripts de prueba envían comandos al servidor Selenium, que a su vez se comunica con el controlador del navegador.
- Una prueba que utiliza Selenium WebDriver implica tres procesos: el script de prueba, el servidor y el navegador.
- El servidor Selenium se encarga de interpretar los comandos de la prueba y enviarlos a uno o más navegadores. La comunicación entre el servidor y el navegador utiliza el Protocolo WebDriver Wire , un protocolo JSON. El comando es interpretado por el controlador del navegador.
- Con Protractor, el script de prueba se ejecuta utilizando Node.js. Protractor ejecuta un comando adicional antes de realizar cualquier acción en el navegador para asegurar que la aplicación que se está probando se haya estabilizado.

Instalación manual

- Se utiliza npm para instalar globalmente Protractor:
  - npm install -g protractor
- Esto instalará dos herramientas de línea de comandos, protractor y WebDriver-manager, para asegurarse de que está funcionando.
  - protractor –versión
- El WebDriver-Manager es una herramienta de ayuda para obtener fácilmente una instancia de un servidor en ejecución Selenium. Para descargar los binarios necesarios:
  - webdriver-manager update
- Para poner en marcha el servidor:
  - webdriver-manager start
- Las pruebas Protractor enviarán solicitudes a este servidor para controlar un navegador local, el servidor debe estar en funcionamiento durante todo el proceso de pruebas.
- Se puede ver información sobre el estado del servidor en:
  - http://localhost:4444/wd/hub

Configuración y Ejecución

- Se configura un fichero con las pruebas a realizar:

```
// Fichero: e2e.conf.js
  exports.config = {
  framework: 'jasmine',
  seleniumAddress: 'http://localhost:4444/wd/hub',
  specs: ['test/*.e2e.js'],
  multiCapabilities: [
     //{ browserName: 'firefox' },
     { browserName: 'chrome' }
   ]
};
```

- Se lanzan las pruebas (con Selenium Server en ejecución):
  - protractor e2e.conf.js

Configuración y Ejecución

- Se configura un fichero con las pruebas a realizar (protractor.conf.js):

```
const { SpecReporter } = require('jasmine-spec-reporter');
exports.config = {
  allScriptsTimeout: 11000,
  specs: [  './src/**/*.e2e-spec.ts' ],
  capabilities: { 'browserName': 'chrome'  },
  directConnect: true,
  baseUrl: 'http://localhost:4200/',
  framework: 'jasmine',
  jasmineNodeOpts: {
    showColors: true,
    defaultTimeoutInterval: 30000,
    print: function() {}
  },
  onPrepare() {
    require('ts-node').register({
      project: require('path').join(__dirname, './tsconfig.json')
    });
    jasmine.getEnv().addReporter(new SpecReporter({ spec: { displayStacktrace: true } }));
  }
};
```

- Se lanzan las pruebas:
  - ng e2e

Elementos Globales

- browser: Envoltura alrededor de una instancia de WebDriver, utilizado para la navegación y la información de toda la página.
  - El método browser.get carga una página.
  - Protractor espera que Angular esté presente la página, por lo que generará un error si la página que está intentando cargar no contiene la biblioteca Angular.
- element: Función de ayuda para encontrar e interactuar con los elementos DOM de la página que se está probando.
  - La función element busca un elemento en la página.
  - Se requiere un parámetro: una estrategia de localización del elemento dentro de la página.
- by: Colección de estrategias elemento localizador.
  - Por ejemplo, los elementos pueden ser encontrados por el selector CSS, por el ID, por el atributo ng-model, …
- protractor: Espacio de nombres de Angular que envuelve el espacio de nombres WebDriver.
  - Contiene variables y clases estáticas, tales como protractor.Key que se enumera los códigos de teclas especiales del teclado.

Visión de conjunto

- Protractor exporta la función global element, que con un localizador devolverá un ElementFinder.
- Esta función recupera un solo elemento, si se necesita recuperar varios elementos, la función element.all obtiene la colección de elementos localizados.
- El ElementFinder tiene un conjunto de métodos de acción, tales como click(), getText() y sendKeys().
- Esta es la forma principal para interactuar con un elemento (etiqueta) de la página y obtener información de respuesta de el.
- Cuando se buscan elementos en Protractor todas las acciones son asíncronas:
  - Por debajo, todas las acciones se envían al navegador mediante el protocolo SON Webdriver Wire Protocol.
  - El navegador realiza la acción tal y como un usuario lo haría de forma nativa o manual.

Localizadores

- Un localizador de Protractor dice cómo encontrar un cierto elemento DOM.
- Los localizadores más comunes son:
  - by.css('.myclass')
  - by.id('myid')
  - by.cssContainingText('a', 'Home')
  - by.xpath('//table/tr/th')
- Los localizadores se pasan a la función element:
  - var tag = element(by.css('some-css'));
  - var arr = element.all(by.css('some-css'));
- Aunque existe una notación abreviada para CSS similar a jQuery:
  - var tag = $('some-css');
- Para encontrar subelementos o listas de subelementos:
  - var uno = element(by.css('some-css')).element(by.tagName('tag-within-css'));
  - var varios = element(by.css('some-css')).all(by.tagName('tag-within-css'));

Localizadores

- Un localizador de Protractor dice cómo encontrar un cierto elemento DOM.
- Los localizadores más comunes son:
  - by.css('.myclass')
  - by.id('myid')
  - by.cssContainingText('a', 'Home')
  - by.xpath('//table/tr/th')
- Los localizadores se pasan a la función element:
  - var tag = element(by.css('some-css'));
  - var arr = element.all(by.css('some-css'));
- Aunque existe una notación abreviada para CSS similar a jQuery:
  - var tag = $('some-css');
- Para encontrar subelementos o listas de subelementos:
  - var uno = element(by.css('some-css')).element(by.tagName('tag-within-css'));
  - var varios = element(by.css('some-css')).all(by.tagName('tag-within-css'));

Localizadores

- by.css
- by.cssContainingText
- by.xpath
- Atajos:
  - by.id('myid') → by.css('#myid')
  - by.className('myclass') → by.css('.myclass')
  - by.tagName('table') → by.css('table')
  - by.name('nombre') → by.css('[name="nombre"]')
  - by.buttonText('Volver') → by.css('input[value="Volver"]')
    by.cssContainingText('button', 'Volver')
  - by.partialButtonText
  - by.linkText('Home') → by.cssContainingText('a', 'Home')
  - by.partialLinkText
- Solo aplicables a Angular JS:
  - binding, exactBinding, model, options, repeater, exactRepeater

Localizadores personalizados

- by.js: evalúa una función en el contexto de la página y devuelve uno o varios elementos.

```
by.js(function() {
  var spans = document.querySelectorAll('span');
  for (var i = 0; i < spans.length; ++i) {
    if (spans[i].offsetWidth > 100) { return spans[i]; }
  }
})
```

- by.addLocator: añade un nuevo localizador a la función by.

```
by.addLocator('buttonTextSimple',
  function(buttonText, opt_parentElement, opt_rootSelector) {
    var using = opt_parentElement || document,
    var buttons = using.querySelectorAll('button');
      return Array.prototype.filter.call(buttons, function(button) {
                return button.textContent === buttonText;
      });
  });
element(by.buttonTextSimple('Volver')).click();
```


Evaluar expresiones (Chrome DevTools)

- La consola de DevTools de Chrome permite conocer el estado de los elementos en una página y de una forma ad hoc.
  - $(): Muestra el primer elemento que coincide con el selector de CSS especificado. Atajo para document.querySelector().
    - $('code') // Returns the first code element in the document.
  - $$(): Muestra un conjunto de todos los elementos que coinciden con el selector de CSS especificado. Alias para document.querySelectorAll().
    - $$('figure') // Returns an array of all figure elements in the document.
  - $x(): Muestra un conjunto de elementos que coinciden con XPath especificada.
    - $x('html/body/p') // Returns an array of all paragraphs in the document body.
- Permite probar las expresiones en caliente.

ElementFinder

- La función element() devuelve un objeto ElementFinder.
- El ElementFinder sabe cómo localizar el elemento DOM utilizando el localizador que se pasa como un parámetro, pero en realidad no lo ha hecho todavía.
- No va a ponerse en contacto con el navegador hasta que un método de acción sea llamado.
- ElementFinder permite invocar acciones como si se produjesen directamente en el navegador.
- Dado que todas las acciones son asíncronas, todos los métodos de acción devuelven una promesa.
- Las acciones sucesivas se encolan y se mandan la navegador ordenadamente.
- Para acciones que deban esperar se usan las promesas:

```
element(by.model('nombre')).getText().then(function(text) {
  expect(text).toBe("MUNDO");
});
```


Promise Pattern

- El Promise Pattern es un patrón de organización de código que permite encadenar llamadas a métodos que se ejecutaran a la conclusión del anterior (flujos).
- Simplifica y soluciona los problemas comunes con el patrón Callback:
  - Llamadas anidadas
  - Complejidad de código

  o.m(1, 2, f(m1(3, f1(4,5,ff(8))) → o.m(1, 2).f().m1(3).f1(4, 5).ff(8)
- Aunque se utiliza extensamente para las operaciones asíncronas, no es exclusivo de las mismas.
- El servicio $q es un servicio de AngularJS que contiene toda la funcionalidad de las promesas (está basado en la implementación Q de Kris Kowal).
- La librería JQuery incluye el objeto $.Deferred desde la versión 1.5.
- Las promesas se han incorporado a los objetos estándar de JavaScript en la versión 6.

Objeto Promise

- Una “promesa” es un objeto que actúa como proxy en los casos en los que no se puede utilizar el verdadero valor porque aún no se conoce (no se ha generado, llegado, …) pero se debe continuar sin esperar a que este disponible (no se puede bloquear la función esperando a su obtención).
- Una “promesa” puede tener los siguientes estados:
  - Pendiente: Aún no se sabe si se podrá o no obtener el resultado.
  - Resuelta: Se ha podido obtener el resultado (Promise.resolve())
  - Rechazada: Ha habido algún tipo de error y no se ha podido obtener el resultado (Promise.reject())
- Los métodos del objeto promesa devuelven al propio objeto para permitir apilar llamadas sucesivas.
- Como objeto, la promesa se puede almacenar en una variable, pasar como parámetro o devolver desde una función, lo que permite aplicar los métodos en distintos puntos del código.

Crear promesas (ES2015)

- El objeto Promise gestiona la creación de la promesa y los cambios de estados de la misma.

```
list() {
  return new Promise((resolve, reject) => {
   this.http.get(this.baseUrl).subscribe( data => resolve(data),  err => reject(err) )
  });
 }
```

- Para crear promesas ya concluidas:
  - Promise.reject: Crea una promesa nueva como rechazada cuyo resultado es igual que el argumento pasado.
  - Promise.resolve: Crea una promesa nueva como resuelta cuyo resultado es igual que su argumento.
- Un Observable se puede convertir en una promesa:

```
import 'rxjs/add/operator/toPromise';
list() {
  return this.http.get(this.baseUrl).toPromise();
 }
```

Invocar promesas

- El objeto Promise creado expone los métodos:
  - then(fnResuelta, fnRechazada): Recibe como parámetro la función a ejecutar cuando termine la anterior y, opcionalmente, la función a ejecutar en caso de que falle la anterior.
  - catch(fnError): Recibe como parámetro la función a ejecutar en caso de que falle.

    list().then(calcular, ponError).then(guardar)
- Otras formas de crear e invocar promesas son:
  - Promise.all: Combina dos o más promesas y realiza la devolución solo cuando todas las promesas especificadas se completan o alguna se rechaza.
  - Promise.race: Crea una nueva promesa que resolverá o rechazará con el mismo valor de resultado que la primera promesa que se va resolver o rechazar entre los argumentos pasados.

Flujo de control

- Las API de WebDriverJS (y, por lo tanto, Protractor) son completamente asíncronas. Todas las funciones devuelven promesas y WebDriverJS mantiene una cola de promesas pendientes, denominada flujo de control, para mantener la ejecución organizada.
- Las promesas no se materializan hasta que es necesario: evaluar una expectativa.
- Protractor adapta Jasmine para que cada especificación espere automáticamente hasta que el flujo de control esté vacío antes de salir.
- Las expectativas de Jasmine también se adaptan para entender las promesas. Por eso, cuando se establece una expectativa, el código en realidad agrega una tarea de expectativa al flujo de control, que se ejecutará después de las otras tareas. Las “suites” (describe) y las especificaciones (it) no emitirán el resultado hasta la conclusión del flujo.

async/await

- El flujo de control del controlador web se eliminará en un futuro, en su lugar se puede sincronizar los comandos con el encadenamiento promesa o con sintaxis async/await del ES7 (ya soportado por TypeScript y NodeJS).
- La sintaxis async/await permite realizar el tratamiento asíncrono de una manera similar al secuencial sin necesidad de usar callback.
- Una función, en cuyo código se utilicen promesas que haya que esperar, debe estar marcada con async. Para esperar la resolución de promesa antes de continuar se marcan las expresiones con await.

```
it('Prueba asíncrona', async () => {
  await browser.get(browser.baseUrl);
  await element(by.css('#nombre')).sendKeys('Mundo');
  expect(await element(by.css('app-root h1')).getText()).toEqual('Hola Mundo!');
 });
```

- De momento, si se quiere utilizar la sintaxis async/await, se debe configurar en protractor.conf.js:
  - SELENIUM_PROMISE_MANAGER: false

La prueba

```
describe('Primera prueba con Protractor', function() {
  it('introducir nombre y saludar', function() {
    browser.get('http://localhost:4200/');
    var txt = element(by.model('vm.nombre'));
    txt.clear();
    txt.sendKeys('Mundo');
    browser.sleep(5000);
    element(by.id('btnSaluda')).click();
    expect(element(by.binding('vm.msg')).getText()).
       toEqual('Hola Mundo');
    browser.sleep(5000);
  });
});
```

Patrón Page Objects

- https://martinfowler.com/bliki/PageObject.html
- Cuando se escriben pruebas de una página web, hay que acceder a los elementos dentro de esa página web para hacer clic en los elementos, teclear entradas y determinar lo que se muestra.
- Sin embargo, si se escriben pruebas que manipulan los elementos HTML directamente, las pruebas serán frágiles ante los cambios en la interfaz de usuario.
- Un objeto de página envuelve una página HTML, o un fragmento, con una API específica de la aplicación, lo que permite manipular los elementos de la página sin excavar en el HTML.

Patrón Page Objects

- La regla básica para un objeto de página es que debe permitir que un cliente de software haga cualquier cosa y vea todo lo que un humano puede hacer.
- El objeto de página debe proporcionar una interfaz que sea fácil de programar y oculta en la ventana.
- Entonces, para acceder a un campo de texto, debe tener métodos de acceso que tomen y devuelvan una cadena, las casillas de verificación deben usar valores booleanos y los botones deben estar representados por nombres de métodos orientados a la acción.
- El objeto de la página debe encapsular los mecanismos necesarios para encontrar y manipular los datos en el propio control gui.
- A pesar del término objeto de "página", estos objetos no deberían construirse para cada página, sino para los elementos significativos en una página.
- Los problemas de concurrencia y asincronía son otro tema que un objeto de página puede encapsular.

Ventajas del patrón Page Objects

- De acuerdo con patrón Page Object, deberíamos mantener nuestras pruebas y localizadores de elementos por separado, esto mantendrá el código limpio y fácil de entender y mantener.
- El enfoque Page Object hace que el programador de marcos de automatización de pruebas sea más fácil, duradero y completo.
- Otra ventaja importante es que nuestro repositorio de objetos de página es independiente de las pruebas de automatización . Mantener un repositorio separado para los objetos de la página nos ayuda a usar este repositorio para diferentes propósitos con diferentes marcos como, podemos integrar este repositorio con otras herramientas como JUnit / NUnit / PhpUnit , así como con TestNG / Cucumber / etc.
- Los casos de prueba se vuelven cortos y optimizados, ya que podemos reutilizar los métodos de objetos de página.
- Los casos de prueba se centran solamente en el comportamiento.
- Cualquier cambio en la IU se puede implementar, actualizar y mantener fácilmente en los objetos y clases de página sin afectar a los casos de pruebas que no estén implicados.

Control del Navegador

- **get**: Navegar hasta el destino especificado y carga los módulos simulados antes de Angular.
- **refresh**: Hace una recarga completa de la página actual y carga módulos simulados antes de Angular.
- **restart**: Reinicia el navegador.
- **restartSync**: Como restart, pero en lugar de devolver una promesa que se resuelve a la nueva instancia del navegador, devuelve la nueva instancia del navegador directamente.
- **wait**: Programa un comando para esperar a que se cumpla una condición o promesa se resuelva.
- **sleep**: Programa un comando para hacer que el proceso duerma durante un período de tiempo determinado.
- **forkNewDriverInstance**: Crea otra instancia de navegador para usar en pruebas interactivas.
- **debugger**: Agrega una tarea al flujo de control para pausar la prueba e inyectar las funciones de ayuda en el navegador, de modo que la depuración se pueda realizar en la consola del navegador.

Control del Navegador

- **setLocation**: Navegua a otra página usando la navegación en la página.
- **getCurrentUrl**: Programa un comando para recuperar la URL de la página actual.
- **getTitle**: Programa un comando para recuperar el título de la página actual.
- **findElement**: Espera a que Angular termine de renderizarse antes de buscar elementos.
- **findElements**: Espera a que Angular termine de renderizarse antes de buscar elementos.
- **isElementPresent**: Comprueba si un elemento está presente en la página.

Control del Navegador

- **actions**: Crea una secuencia de acciones de usuario utilizando:
  - click, clickAndHold, contextClick, doubleClick, dragAndDrop, release, moveByOffset, moveToElement, keyDown, keyUp, sendKeys
  - pause, build, perform.
- **touchActions**: Crea una nueva secuencia táctil utilizando este controlador.
- **executeScript**: Programa un comando para ejecutar JavaScript en el contexto del marco o ventana actualmente seleccionado.
- **executeAsyncScript**: Programa un comando para ejecutar JavaScript asíncrono en el contexto del marco o ventana actualmente seleccionado.
- **call**: Programa un comando para ejecutar una función personalizada dentro del contexto del flujo de control de webdriver.
- **close**: Programa un comando para cerrar la ventana actual.
- **switchTo**: Se utiliza para cambiar el enfoque de WebDriver a un marco o ventana
- **getPageSource**: Programa un comando para recuperar el fuente de la página actual.
- **takeScreenshot**: Programa un comando para tomar una captura de pantalla.

Condiciones de espera

- browser.wait(condición, timeout de espera, mensaje si expira)
  - **alertIsPresent**: Esperar una alerta para estar presente.
  - **elementToBeClickable**: Una Expectativa para verificar un elemento está visible y habilitado, de modo que puede hacer clic en él.
  - **textToBePresentInElement**: Una expectativa para verificar si el texto dado está presente en el elemento.
  - **textToBePresentInElementValue**: Una expectativa para verificar si el texto dado está presente en el valor del elemento.
  - **titleContains**: Una expectativa para comprobar que el título contiene una subcadena que distingue entre mayúsculas y minúsculas.
  - **titleIs**: Una expectativa para comprobar el título de una página.
  - **urlContains**: Una expectativa para comprobar que la URL contiene una subcadena que distingue entre mayúsculas y minúsculas.
  - **urlIs**: Una expectativa para comprobar la URL de una página.
  - **presenceOf**: Una expectativa para verificar que un elemento esté presente en el DOM de una página.
  - **stalenessOf**: Una expectativa para verificar que un elemento no esté presente al DOM de una página.
  - **visibilityOf**: Una expectativa para verificar que un elemento esté presente en el DOM de una página y sea visible.
  - **invisibilityOf**: Una expectativa para verificar que un elemento sea invisible o no esté presente en el DOM.
  - **elementToBeSelected**: Una expectativa para comprobar que una selección esta seleccionada.
  - **not**: Niega el resultado de una promesa.
  - **and**: Espera a que se cumplan todas las condiciones o que se evalúe una como falsa.
  - **or**: Espera a que se cumplan una de varias condiciones.

Colecciones de elementos

- **$$:** Función abreviada para encontrar matrices de elementos por css.
- **all**: Las llamadas a ElementArrayFinder se pueden encadenar para encontrar una matriz de elementos utilizando los elementos actuales en este ElementArrayFinder como punto de partida.
- **count**: Cuenta el número de elementos representados por el ElementArrayFinder.
- **isPresent**: Devuelve verdadero si hay elementos presentes que coincidan con el buscador.
- **get**: Obtiene un elemento dentro del ElementArrayFinder por índice.
- **first**: Obtiene el primer elemento coincidente para el ElementArrayFinder.
- **last**: Obtiene el último elemento coincidente para el ElementArrayFinder.
- **locator**: Devuelve el localizador más relevante.
- **then**: Resuelve la promesa.
- **filter**: Aplica una función de filtro a cada elemento dentro de ElementArrayFinder.
- **each**: Llama a la función de entrada para cada ElementFinder representado por ElementArrayFinder.
- **map**: Aplica una función de mapa a cada elemento dentro de ElementArrayFinder.
- **reduce**: Aplica una función de reducción contra un acumulador y cada elemento encontrado usando el localizador (de izquierda a derecha).
- **clone**: Crea una copia superficial de ElementArrayFinder.
- **evaluate**: Evalúa la entrada como si estuviera en el alcance de los elementos subyacentes actuales (js).
- **allowAnimations**: Determina si la animación está permitida en los elementos subyacentes actuales.

Operara sobre el elemento

- **getId**: Obtiene la representación de cadena de ID de WebDriver para este elemento web.
- **getDriver**: Obtiene el elemento contenedor de este elemento web.
- **getTagName**: Obtiene la etiqueta / nombre de nodo de este elemento.
- **getCssValue**: Obtiene el estilo calculado de un elemento.
- **getAttribute**: Programa un comando para consultar el valor del atributo dado del elemento.
- **getText**: Obtiene el texto interior visible de este elemento, incluidos los subelementos, sin espacios en blanco iniciales ni finales.
- **getSize**: Programa un comando para calcular el tamaño del cuadro delimitador de este elemento, en píxeles.
- **getLocation**: Programa un comando para calcular la ubicación de este elemento en el espacio de la página.
- **isEnabled**: Programa un comando para consultar si el elemento DOM representado por esta instancia está habilitado, como lo indica el atributo disabled.
- **isSelected**: Programa un comando para consultar si este elemento está seleccionado.

Operara sobre el elemento

- **isDisplayed**: Programa un comando para probar si este elemento se muestra actualmente.
- **isPresent**: Determina si el elemento localizado está presente en la página.
- **isElementPresent**: Determina mediante un localizador si un elemento está presente en la página.
- **click**: Programa un comando para hacer clic en este elemento.
- **sendKeys**: Programa un comando para escribir una secuencia en el elemento DOM representado por esta instancia.
- **submit**: Programa un comando para enviar el formulario que contiene este elemento (o este elemento si es un elemento FORM).
- **clear**: Programa un comando para borrar el value de este elemento.
- **equals**: Compara un elemento con este por la igualdad.
- **clone**: Crear una copia superficial de ElementFinder.

Operara sobre el elemento

- **all**: Las llamadas a all pueden estar encadenadas para encontrar una matriz de elementos dentro de un padre.
- **element**: Las llamadas a element pueden estar encadenadas para encontrar elementos dentro de un padre.
- **$$:** Las llamadas a $$ pueden estar encadenadas para encontrar una matriz de elementos dentro de un padre.
- **$:** Las llamadas a $ pueden estar encadenadas para encontrar elementos dentro de un padre.
- **evaluate**: Evalúa la entrada dentro del alcance del elemento actual (js).
- **allowAnimations**: Determina si la animación está permitida en los elementos subyacentes actuales.
- **locator**: Devuelve el localizador más relevante.
- **getWebElement**: Devuelve el elemento web representado por este ElementFinder.
- **takeScreenshot**: Toma una captura de pantalla de la región visible que abarca el rectángulo delimitador de este elemento.

Depuración del caso de prueba

- Con Chrome:
  - Lanzar la prueba manualmente:
    - node --inspect-brk node_modules/protractor/bin/protractor e2e/protractor.conf.js
  - Inspeccionar y depurar en el navegador:
    - chrome://inspect
- Con Visual Studio Code:
  - Configurar el depurador (entorno NodeJS) en el archivo .vscode/launch.json:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "Depurar e2e",
      "program": "${workspaceRoot}/node_modules/protractor/bin/protractor",
      "args": ["${workspaceRoot}/e2e/protractor.conf.js"],
      "outFiles": [  "${workspaceFolder}/dist/out-tsc/**/*.js“ ]
    }
  ]
}
```

  - Poner puntos de ruptura y ejecutar las pruebas con el entorno

Organizar pruebas

- Es posible separar sus pruebas en varios conjuntos de pruebas. En el archivo de configuración, se puede configurar la opción de suites:

```
suites: {
  homepage: './src/homepage/**/*.e2e-spec.ts',
  search: [
    './src/contact_search/**/*.e2e-spec.ts',
    './src/venue_search/**/*.e2e-spec.ts'
  ]
},
```

- Desde la línea de comandos, se puede cambiar fácilmente entre ejecutar una, varias (separadas por comas) o todas las series de pruebas.
  - protractor protractor.conf.js --suite homepage
  - protractor protractor.conf.js --suite homepage,search
  - ng e2e --suite homepage

Backend Mock

- Siguiendo la misma regla de oro que en las pruebas unitarias, las pruebas e2e deben estar aisladas de sus dependencias salvo cuando se estén probando dichas dependencias. Así mismo, el resultado de las pruebas debe ser previsible.
- La dependencia fundamental de las aplicaciones SPA es el API Rest con el back-end del servidor.
- Eso quiere decir que se necesitan dobles de pruebas para dependencias API y hay que reemplazarlas por mocks: API Mocking.
- Entre las ventajas de esta aproximación se encuentran:
  - Devuelven resultados determinísticos
  - Permiten crear o reproducir determinados estados (por ejemplo errores de conexión)
  - Obtienen resultados mucho mas rápidamente y a menor coste, incluso offline.
  - Permiten el inicio temprano de las pruebas incluso cuando las APIs todavía no están disponibles.
  - Permiten incluir atributos o métodos exclusivamente para el testeo.

API Mocking

- La solución pasa por crear un servidor de mocks, esto significa que el servidor imitará el comportamiento de nuestro servidor real, devolviendo datos de prueba o datos esperados tras las peticiones que quiero poner a prueba.
- Hay muchos frameworks destinados a la creación de servidores de mocks, como el apropiadamente nombrado MockServer, WireMock o Sandbox. Otros son más simples y facilitan el proceso, permitiendo indicar la respuesta a dar directamente, como Mocky o MockBin. Incluso, la herramienta de desarrollo de API Postman ofrece un servidor de mocks integrado.
- Eso sí, hay que tener en cuenta que es muy importante que la API esté bien diseñada y documentada, ya que si hay errores en la especificación, habrá disparidad en el comportamiento de los mocks, causando que el frontend no termine de encajar cuando se haga el cambio al backend real.
- Los mocks de API son una herramienta muy potente que permite desarrollar y probar el front-end como un componente independiente del back-end, facilitando y reduciendo tiempos de desarrollo, lo que se traduce en el aumento de la productividad del equipo.
- La virtualización de API es el proceso de usar una herramienta que crea una copia virtual de su API, que refleja todas las especificaciones de su API de producción, y usar esta copia virtual en lugar de su API de producción para realizar pruebas.

Proxy a un servidor de back-end

- Se puede usar el soporte de proxy en el servidor de desarrollo del webpack para desviar ciertas URL a un servidor mock backend.
- Crea un archivo proxy.conf.json junto a angular.json.

```
{
  "/api": {
    "target": "http://localhost:4321",
    "pathRewrite": { "^/api": "/ws" },
    "secure": false,
    "logLevel": "debug"
  }
}
```

- Cambiar la configuración del Angular CLI en angular.json:

```
"architect": {
  "serve": {
    "builder": "@angular-devkit/build-angular:dev-server",
    "options": {
      …
      "proxyConfig": "proxy.conf.json"
    },
```

Configurar el navegador: Firefox

- Instalación de dirvers:
  - node bin/webdriver-manager update --help
  - node bin/webdriver-manager update
  - node bin/webdriver-manager update --gecko
- Configuración:

```
capabilities: {
  browserName: 'firefox',
  "moz:firefoxOptions": {
    "log": {"level": "trace"}
  },
 },
 directConnect: true,
```


Configurar el navegador: Edge

- Descargar driver:
  - https://developer.microsoft.com/en-us/microsoft-edge/tools/webdriver/#downloads
- Copiar a la carpeta de dirvers:
  - node_modules\protractor\node_modules\webdriver-manager\selenium
- Configuración:

```
capabilities: {
  'browserName': 'MicrosoftEdge',
  'platform': 'windows',
 },
 directConnect: false,
```

Configurar el navegador: IE

- Instalación de dirvers:
  - node bin/webdriver-manager update –ie64
- Configuración:

```
capabilities: {
  'browserName': 'internet explorer',
  'platform': 'windows',
  'version': '11'
},
directConnect: false,
```


Pruebas contra múltiples navegadores

- Pruebas contra múltiples navegadores

```
multiCapabilities: [{
  'browserName': 'firefox'
}, {
  'browserName': 'chrome'
}]
```

- Usar múltiples navegadores en la misma prueba:

```
it('Prueba en múltiples navegadores', () => {
  browser.get('http://angular.io');
  let browser2 = browser.forkNewDriverInstance();
  browser2.get(browser.baseUrl);
  browser.sleep(500);
  browser2.sleep(500);
});
```

  - Las variables globales element, $, $$ y browser están todos asociados con el navegador por defecto, para acceder a otros: browser2.element(by…)

Selenium

- http://www.seleniumhq.org/
- El Selenium es un conjunto de herramientas para automatizar los navegadores web, robot que simula la interacción del usuario con el navegador, originalmente pensado como entorno de pruebas de software para aplicaciones basadas en la web.
- Como principales herramientas Selenium cuenta con:
  - Selenium IDE: una herramienta para grabar y reproducir secuencias de acciones con el navegador que permite crear pruebas sin usar un lenguaje de scripting para pruebas.
  - Selenium Core: API para escribir pruebas automatizadas y de regresión en un amplio número de lenguajes de programación populares incluyendo Java, C#, Ruby, Groovy, Perl, Php y Python.
  - WebDriver: interfaces que permite ejecutar las pruebas de forma nativa usando la mayoría de los navegadores web modernos en diferentes sistemas operativos como Windows, Linux y OSX.
  - Selenium Grid: Permite ejecutar muchas pruebas de un mismo grupo en paralelo o pruebas en múltiples entornos. Tiene la ventaja que un conjunto de pruebas muy grande puede dividirse en varias maquinas remotas para una ejecución más rápida o si se necesitan repetir las mismas pruebas en múltiples entornos.

Selenium IDE

- Es el entorno de desarrollo integrado para pruebas con Selenium que permite grabar, editar y depurar fácilmente las pruebas.
- Solo está disponible como una extensión de Firefox y Chrome.
- Se pueden desarrollar automáticamente scripts al crear una grabación y de esa manera se puede editar manualmente con sentencias y comandos para que la reproducción de nuestra grabación sea correcta
- Los scripts se generan en un lenguaje de scripting especial para Selenium a menudo denominado Selanese.
- Selanese provee comandos que dicen al Selenium que hacer y pueden ser:
  - **Acciones**: son comandos que generalmente manipulan el estado de la aplicación, ejecutan acciones sobre objetos del navegador, como hacer click en un enlace, escribir en cajas de texto o seleccionar de una lista de opciones. Muchas acciones pueden ser llamadas con el sufijo "AndWait" que indica la acción hará que el navegador realice una llamada al servidor y que se debe esperar a una nueva página se cargue.
  - **Descriptores de acceso**: examinan el estado de la página y almacenan los resultados en variables.
  - **Aserciones**: son como descriptores de acceso, pero las muestras confirman que el estado de la solicitud se ajusta a lo que se esperaba, verifican la presencia de un texto en particular o la existencia de elementos.

Selenium IDE

- Dispone de una selección inteligente de campos usando ID, nombre, Xpath o DOM según se necesite.
- Para la depuración permite la configuración de los puntos de interrupción, iniciar y detener la ejecución de un caso de prueba desde cualquier punto dentro del caso de prueba e inspeccionar la forma en el caso de prueba se comporta en ese punto.
- ~~Permite exportar los casos de prueba a Java, C# y Ruby, actuando como embriones en la creación de los casos de prueba para WebDriver.~~
- Selenium IDE dispone de un amplio conjunto de extensiones adicionales que ayudan o simplifican la elaboración de los casos de pruebas.

Localizadores

- Localizar por Id:
  - id=loginForm
- Localizar por Name
  - name=username
- Localizar por XPath
  - xpath=//form[@id='loginForm']
- Localizar por el texto en los hipervínculos
  - link=Continue
- Localizar por CSS
  - css=input[name="username"]

Variables

- Se puede usar variable en Selenium para almacenar constantes al principio de un script. Además, cuando se combina con un diseño de prueba controlado por datos, las variables de Selenium se pueden usar para almacenar valores pasados a la prueba desde la línea de comandos, desde otro programa o desde un archivo.
  - store target:valor value:varName
- Para acceder al valor de una variable:
  - ${userName}
- Hay métodos diponibles para recuperar información de la página y almacenarla en variables:
  - storeAttribute, storeText, storeValue, storeTitle, storeXpathCount

Afirmar y Verificar

- Una "afirmación" hará fallar la prueba y abortará el caso de prueba actual, mientras que una "verificación" hará fallar la prueba pero continuará ejecutando el caso de prueba.
  - Tiene muy poco sentido para comprobar que el primer párrafo de la página sea el correcto si la prueba ya falló al comprobar que el navegador muestra la página esperada. Por otro lado, es posible que desee comprobar muchos atributos de una página sin abortar el caso de prueba al primer fallo, ya que esto permitirá revisar todas los fallos en la página y tomar la acción apropiada.
- Selenese permite múltiples formas de comprobar los elementos de la interfaz de usuario pero hay que decidir el métodos mas apropiado:
  - ¿Un elemento está presente en algún lugar de la página?
  - ¿El texto especificado está en algún lugar de la página?
  - ¿El texto especificado está en una ubicación específica en la página?
- Métodos:
  - assert, assertAlert, assertChecked, assertNotChecked, assertConfirmation, assertEditable, assertNotEditable, assertElementPresent, assertElementNotPresent, assertPrompt, assertSelectedValue, assertNotSelectedValue, assertSelectedLabel, assertText, assertNotText, assertTitle, assertValue
  - verify, verifyChecked, verifyNotChecked, verifyEditable, verifyNotEditable, verifyElementPresent, verifyElementNotPresent, verifySelectedValue, verifyNotSelectedValue, verifyText, verifyNotText, verifyTitle, verifyValue, verifySelectedLabel

Proceso de Prueba

- Descomponer la aplicación web existente para identificar qué probar
- Identificar con qué navegadores probar
- Elige el mejor lenguaje para ti y tu equipo.
- Configura Selenium para que funcione con cada navegador que te interese.
- Escriba pruebas de Selenium mantenibles y reutilizables que serán compatibles y ejecutables en todos los navegadores.
- Cree un circuito de retroalimentación integrado para automatizar las ejecuciones de prueba y encontrar problemas rápidamente.
- Configura tu propia infraestructura o conéctate a un proveedor en la nube.
- Mejora drásticamente los tiempos de prueba con la paralelización
- Mantente actualizado en el mundo Selenium.

WebDriver

```
@BeforeClass
public static void setUpClass() throws Exception {
        System.setProperty("webdriver.chrome.driver", "C:/Archivos/…/chromedriver.exe");
}
@Before
public void setUp() throws Exception {
  driver = new ChromeDriver();
  baseUrl = "http://localhost/";
  driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);
}
@Test
public void testLoginOK() throws Exception {
  driver.get(baseUrl + "/login.php");
  driver.findElement(By.id("login")).sendKeys("admin");
  driver.findElement(By.id("password")).sendKeys("admin");
  driver.findElement(By.cssSelector("input[type=\"submit\"]")).click();
  try {
    assertEquals("", driver.findElement(By.cssSelector("img[title=\"Main Menu\"]")).getText());
  } catch (Error e) {
    verificationErrors.append(e.toString());
  }
}
```

Maven

```
<dependency>
  <groupId>org.seleniumhq.selenium</groupId>
  <artifactId>selenium-java</artifactId>
  <version>3.13.0</version>
</dependency>
```

Ejecutar en línea de comandos

- Requiere tener instalado NodeJS (https://nodejs.org)
- Instalar CLI
  - npm install -g selenium-side-runner
- Instalar los WebDriver:
  - Crear una carpeta y referenciarla en el PATH del sistema.
  - Descargar los drivers (https://www.seleniumhq.org/download/)
  - Copiarlos a la carpeta creada.
- Para ejecutar las suites de pruebas:
  - selenium-side-runner project.side project2.side *.side
- Para ejecutar en diferentes navegadores:
  - selenium-side-runner *.side -c "browserName=Chrome"
  - selenium-side-runner *.side -c "browserName=firefox"

Pruebas de Aceptación: BDD

- El Desarrollo Dirigido por Comportamiento (Behaviour Driver Development) es una evolución de TDD (Test Driven Development o Desarrollo Dirigido por Pruebas), el concepto de BDD fue inicialmente introducido por Dan North como respuesta a los problemas que surgían al enseñar TDD.
- En BDD también vamos a escribir las pruebas antes de escribir el código fuente, pero en lugar de pruebas unitarias, lo que haremos será escribir pruebas que verifiquen que el comportamiento del código es correcto desde el punto de vista de negocio. Tras escribir las pruebas escribimos el código fuente de la funcionalidad que haga que estas pruebas pasen correctamente. Después refactorizamos el código fuente.
- Partiremos de historias de usuario, siguiendo el modelo “Como [rol] quiero [característica] para [los beneficios]”. A partir de aquí, en lugar de describir en 'lenguaje natural' lo que tiene que hacer esa nueva funcionalidad, vamos a usar un lenguaje ubicuo (un lenguaje semiformal que es compartido tanto por desarrolladores como personal no técnico) que nos va a permitir describir todas nuestras funcionalidades de una única forma.

Gherkin

- Gherkin, es un lenguaje comprensible por humanos y por ordenadores, con el que vamos a describir las funcionalidades, definiendo el comportamiento del software, sin entrar en su implementación. Se trata de un lenguaje fácil de leer, fácil de entender y fácil de escribir. Es un lenguaje de los que Martin Fowler llama 'Business Readable DSL', es decir, 'Lenguaje Específico de Dominio legible por Negocio'.
- Utilizar Gherkin nos va a permitir crear una documentación viva a la vez que automatizamos los tests, haciéndolo además con un lenguaje que puede entender negocio.
- Para empezar a hacer BDD sólo nos hace falta conocer 5 palabras, con las que construiremos sentencias con las que vamos a describir las funcionalidades:

Gherkin

- Feature (característica): Indica el nombre de la funcionalidad que vamos a probar. Debe ser un título claro y explícito. Incluimos aquí una descripción en forma de historia de usuario: “Como [rol] quiero [característica] para [los beneficios]”. Sobre esta descripción empezaremos a construir nuestros escenarios de prueba.
- Scenario: Describe cada escenario que vamos a probar.
- Given (dado): Provee el contexto para el escenario en que se va a ejecutar el test, tales como el punto donde se ejecuta el test, o prerequisitos en los datos. Incluye los pasos necesarios para poner al sistema en el estado que se desea probar.
- When (cuando): Especifica el conjunto de acciones que lanzan el test. La interacción del usuario que acciona la funcionalidad que deseamos testear.
- Then (entonces): Especifica el resultado esperado en el test. Observamos los cambios en el sistema y vemos si son los deseados.

Cucumber

- Cucumber es una de las herramientas que podemos utilizar para automatizar nuestras pruebas en BDD. Cucumber nos va permitir ejecutar descripciones funcionales en Gherkin como pruebas de software automatizadas.
- Cucumber fue creada en 2008 por Aslak Hellesoy y está escrito en Ruby, aunque tiene implementaciones para casi cualquier lenguaje de programación: JRuby (usando Cucumber-JVM), Java, Groovy, JavaScript, JavaScript (usando Cucumber-JVM y Rhino), Clojure, Gosu, Lua, .NET (usando SpecFlow), PHP (usando Behat), Jython, C++ o Tcl.
- Cucumer es probablemente la herramienta más conocida y más utilizada para automatizar pruebas en BDD, pero existen otros frameworks con los que poder hacer BDD para los lenguajes de programación más habituales.

Cucumber

- Instalación:
  - npm install --save-dev cucumber protractor-cucumber-framework chai chai-as-promised
- Crear directorios:
  - acceptance
    - features
    - steps
- Crear fichero de configuración acceptance/protractor.conf.js
- Crear ficheros de features y steps
- Ejecutar la prueba:
  - node node_modules/protractor/bin/protractor acceptance/protractor.conf.js

protractor.conf.js

```
module.exports.config = {
   framework: 'custom',
   frameworkPath: 'node_modules/protractor-cucumber-framework',
   cucumberOpts: {
      require: ['steps/*.steps.js'],
      strict: true
   },
   specs: ['features/*.feature'],
   capabilities: {
      browserName: 'chrome',
   }
};
```


example.feature

```
Feature: Example
  Scenario: should navigate to the main page
    When I navigate to "https://angular.io/"
    Then the title should be "Angular"

  Scenario: should be able to see the Docs page
    When I navigate to "https://angular.io/"
    When I click the Docs button
    Then I should see a "Introduction to the Angular
Docs" article
```

example.steps.js

```
const { When, Then } = require('cucumber');
const expect = require('chai').use(require('chai-as-promised')).expect;

When('I navigate to {string}', function (site) {
  return browser.get(site);
});
Then('the title should be {string}', function (title) {
  return expect(browser.getTitle()).to.eventually.eql(title);
});
When('I click the Docs button', function () {
  return element(by.css('[title="Docs"]')).click();
});
Then('I should see a {string} article', function (title) {
  return expect(element(by.id('introduction-to-the-angular-
docs')).getText()).to.eventually.eql(title);
});
```