

## 7 Principios del Testing

1

La prueba muestra la presencia de defectos, no su ausencia.



La prueba puede mostrar la presencia de defectos, pero no puede probar que no hay defectos. La prueba reduce la probabilidad de que queden defectos no descubiertos en el software pero, incluso si no se encuentran defectos, el proceso de prueba no es una demostración de corrección.

2

La prueba exhaustiva es imposible.



No es posible probar todo —todas las combinaciones de entradas y precondiciones— excepto en casos triviales. En lugar de intentar realizar pruebas exhaustivas se deberían utilizar el análisis de riesgos, las técnicas de prueba y las prioridades para centrar los esfuerzos de prueba.

3

La prueba temprana ahorra tiempo y dinero.



Para detectar defectos de forma temprana, las actividades de testing tanto estáticas como dinámicas deben iniciarse lo antes posible en el ciclo de vida de desarrollo de software. La prueba temprana a veces se denomina desplazamiento hacia la izquierda. La prueba temprana en el ciclo de vida de desarrollo de software ayuda a reducir o eliminar cambios costosos.

4

Los defectos se agrupan.



En general, un pequeño número de módulos contiene la mayoría de los defectos descubiertos durante la prueba previa al lanzamiento, o es responsable de la mayoría de los fallos operativos. Las agrupaciones de defectos previstas y las agrupaciones de defectos reales observadas en la prueba o producción son una aportación importante a un análisis de riesgos utilizado para centrar el esfuerzo de la prueba.

# 5

Cuidado con la paradoja del pesticida.



Si las mismas pruebas se repiten una y otra vez, eventualmente estas pruebas ya no encontrarán ningún defecto nuevo. Para detectar nuevos defectos, es posible que sea necesario cambiar las pruebas y los datos de prueba existentes, y es posible que sea necesario redactar nuevas pruebas. En algunos casos como la prueba de regresión automatizada, la paradoja del pesticida tiene un resultado beneficioso, que es el número relativamente bajo de defectos de regresión.

# 6

La prueba depende del contexto.



La prueba se realiza de manera diferente en diferentes contextos. Por ejemplo, el software de control industrial de seguridad crítica se prueba de forma diferente a una aplicación móvil de comercio electrónico. Como ejemplo adicional, la prueba en un proyecto ágil se realiza de manera diferente a la prueba en un proyecto que se desarrolla según un ciclo de vida secuencial.

# 7

La ausencia de errores es una falacia.



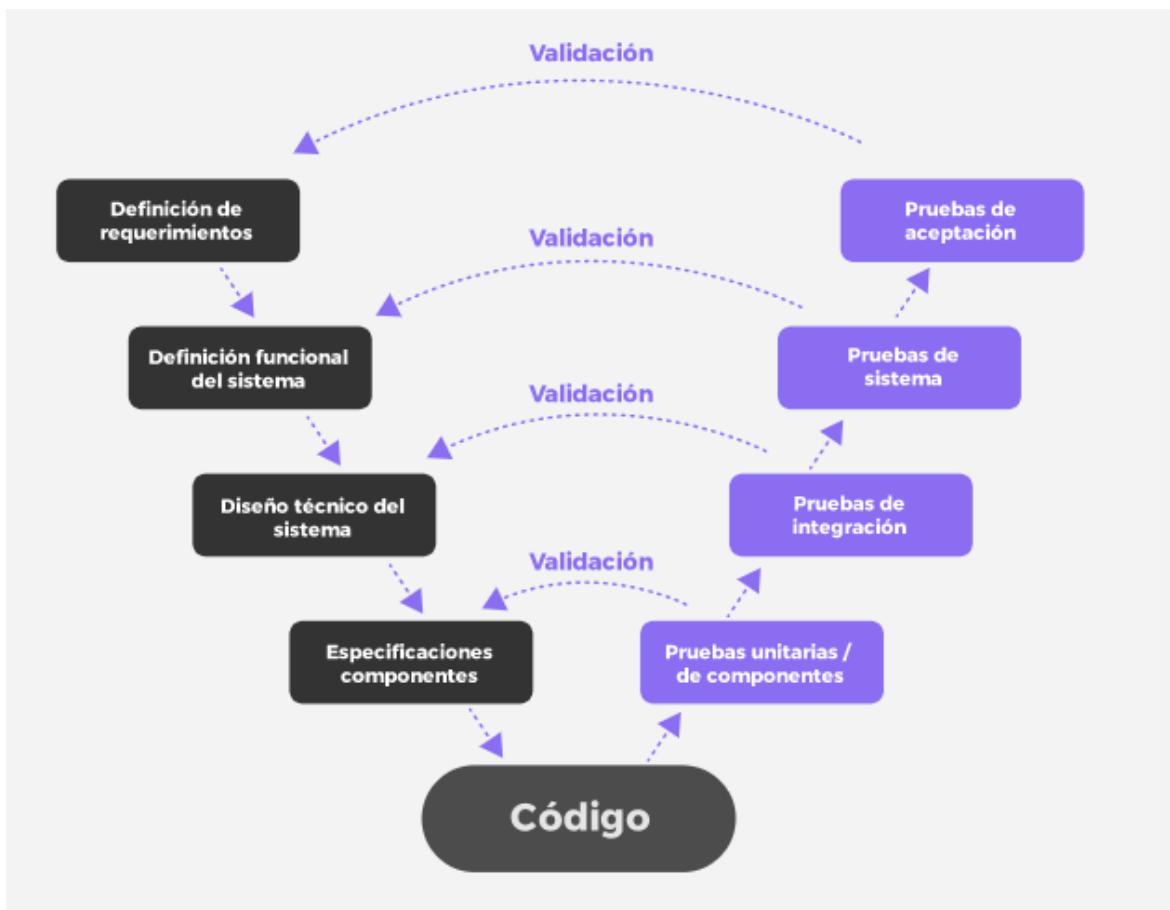
Es una falacia, es decir, es una creencia equivocada, esperar que sólo encontrar y corregir un gran número de defectos asegure el éxito de un sistema. Por ejemplo, la realización de pruebas exhaustivas de todos los requisitos especificados y la reparación de todos los defectos encontrados podría dar lugar a un sistema difícil de utilizar, que no satisface las necesidades y expectativas de los usuarios o que es peor en comparación con otros sistemas de la competencia.

# Niveles de prueba

Los niveles de prueba son grupos de actividades de prueba que se organizan y gestionan conjuntamente. Cada nivel de prueba es una instancia del proceso de prueba, realizadas en relación con el software en un nivel de desarrollo determinado, desde unidades o componentes individuales hasta sistemas completos o, en su caso, sistemas de sistemas. Los niveles de prueba están relacionados con otras actividades dentro del ciclo de vida de desarrollo de software y son:

- Prueba de componente.
- Prueba de integración.
- Prueba de sistema.
- Prueba de aceptación.

Esto se puede ver gráficamente en el modelo V donde para cada etapa de desarrollo existe un nivel de prueba correspondiente y cada nivel de prueba tienen unas características definidas.



Se requiere un entorno de prueba adecuado para cada nivel de prueba. En la prueba de aceptación, por ejemplo, un entorno de prueba similar al de producción es ideal, mientras que en la prueba de componente los desarrolladores suelen utilizar su propio entorno de desarrollo.

## Pruebas de componentes

Las pruebas de componentes son la primera fase de las pruebas dinámicas y se realizan sobre cada módulo o componente del software que se pueda probar de forma independiente. También es conocida como prueba unitaria o de módulo. Siendo un módulo la unidad mínima funcional de un

sistema. Por ejemplo, en una calculadora que realiza las operaciones de suma, resta, multiplicación y división, cada operación estará desarrollada en un módulo.

El objetivo es comprobar que el componente está correctamente codificado soportando el ingreso de datos erróneos o inesperados y demostrando así la capacidad de tratar errores de manera controlada.

En general las lleva a cabo el desarrollador.

La prueba de componente a menudo se realiza de forma aislada del resto del sistema, dependiendo del modelo de ciclo de vida de desarrollo de software y del sistema, lo que puede requerir objetos simulados, virtualización de servicios, arneses, stubs y controladores.

Las pruebas de componentes pueden incluir:

- **Pruebas funcionales** como, por ejemplo, verificar exactitud de cálculos.
- **Pruebas no funcionales** para verificar el comportamiento de los recursos, por ejemplo, fugas de memoria relacionadas al manejo inapropiado de variables, rendimiento o robustez que permitan a la aplicación crecer en volumen o velocidad.

**Pruebas estructurales** como la cobertura de decisiones para garantizar que las diferentes ramas de condiciones estén probadas.

## Pruebas de integración

Las pruebas de integración sirven para asegurar que la comunicación, enlaces y datos compartidos entre módulos o con diferentes partes del sistema —como sistema operativo, sistema de archivos o hardware— funcionan de acuerdo a lo esperado.

Antes de estas pruebas los componentes pasaron las pruebas unitarias, por lo que el enfoque ahora es sobre el flujo de control entre los módulos y sobre los datos que son intercambiados entre ellos de manera independiente.

Hay más de un nivel de prueba de integración dependiendo del tamaño del objeto de prueba:

- **Prueba de integración de componentes:** prueban las interacciones entre los componentes del software. Se realizan después de las pruebas de componentes y son generalmente automatizadas. En el desarrollo iterativo e incremental, la prueba de integración de componentes suele formar parte del proceso de integración continua.
- **Pruebas de integración del sistema:** prueban las interacciones entre diferentes sistemas, paquetes y servicios. Pueden realizarse después de la prueba de sistema o en paralelo con las actividades de prueba de sistema en curso. También pueden cubrir las interacciones con interfaces proporcionadas por organizaciones externas —por ejemplo, servicios web—.

La prueba de integración de componentes suele ser responsabilidad de los desarrolladores. La prueba de integración de sistemas es, en general, responsabilidad de los testers. En condiciones ideales, los testers que realizan la prueba de integración de sistemas deberían entender la arquitectura del sistema y deberían haber influido en la planificación de la integración.

La prueba de integración de componentes y la prueba de integración de sistemas deben concentrarse en la integración propiamente dicha. Por ejemplo, si se integra el módulo A con el módulo B, la prueba debe centrarse en la comunicación entre los módulos, no en la funcionalidad de los módulos individuales, como debería haberse hecho durante la prueba de componente.

Se puede utilizar los tipos de prueba funcional, no funcional y estructural.

## Pruebas de Sistemas

La prueba de sistema se centra en el comportamiento y las capacidades de todo un sistema o producto, a menudo teniendo en cuenta las tareas extremo a extremo que el sistema puede realizar y los comportamientos no funcionales que exhibe mientras realiza esas tareas.

A menudo, la prueba de sistema produce información que es utilizada por los implicados para tomar decisiones con respecto al lanzamiento. La prueba de sistema también puede satisfacer requisitos o estándares legales o regulatorios. El entorno de prueba debe corresponder, en condiciones ideales, al entorno objetivo final o entorno de producción.

La prueba de sistema debe utilizar las técnicas más apropiadas para los aspectos del sistema que serán probados. Por ejemplo, se puede crear una tabla de decisión para verificar si el comportamiento funcional es el descrito en términos de reglas de negocio.

Los testers independientes, en general, llevan a cabo la prueba de sistema. Los defectos en las especificaciones —por ejemplo, la falta de historias de usuario, los requisitos de negocio establecidos de forma incorrecta, etc.— pueden llevar a una falta de comprensión o a desacuerdos sobre el comportamiento esperado del sistema. Tales situaciones pueden causar falsos positivos y falsos negativos, lo que hace perder tiempo y reduce la eficacia de la detección de defectos, respectivamente. La participación temprana de los testers en el perfeccionamiento de la historia de usuario o en actividades de prueba estática, como las revisiones, ayuda a reducir la incidencia de tales situaciones.

## Pruebas de aceptación

La prueba de aceptación se centra normalmente en el comportamiento y las capacidades de todo un sistema o producto. Puede producir información para evaluar el grado de preparación del sistema para su despliegue y uso por parte del cliente —usuario final—. Los defectos pueden encontrarse durante las pruebas de aceptación, pero encontrar defectos no suele ser un objetivo, y encontrar un número significativo de defectos durante la prueba de aceptación puede, en algunos casos, considerarse un riesgo importante para el proyecto. La prueba de aceptación también puede satisfacer requisitos o normas legales o reglamentarias.

Las formas comunes de prueba de aceptación incluyen las siguientes:

- **Prueba de aceptación de usuario:** se centra normalmente en la validación de la idoneidad para el uso del sistema por parte de los usuarios previstos en un entorno operativo real o simulado. El objetivo principal es crear confianza en que los usuarios pueden utilizar el sistema para satisfacer sus necesidades, cumplir con los requisitos y realizar los procesos de negocio con la mínima dificultad, coste y riesgo.
- **Prueba de aceptación operativa:** realizada por parte del personal de operaciones o de la administración del sistema se lleva a cabo, por lo general, en un entorno de producción (simulado). La prueba se centra en los aspectos operativos, y pueden incluir:

- Prueba de copia de seguridad y restauración.
- Instalación, desinstalación y actualización.
- Recuperación ante desastres.
- Gestión de usuarios.
- Tareas de mantenimiento.
- Carga de datos y tareas de migración.
- Comprobación de vulnerabilidades de seguridad.
- Prueba de rendimiento.

El objetivo principal es generar confianza en que los operadores o administradores del sistema pueden mantener el sistema funcionando correctamente para los usuarios en el entorno operativo, incluso en condiciones excepcionales o difíciles.

- **Prueba de aceptación contractual y normativa:** se realiza en función de los criterios de aceptación del contrato para el desarrollo de software a medida. Los criterios de aceptación deben definirse cuando las partes acuerdan el contrato. La prueba de aceptación normativa se lleva a cabo con respecto a cualquier norma que deba cumplirse, como las normas gubernamentales, legales o de seguridad física. Suele ser realizada por usuarios o por testers independientes, en ocasiones los resultados son presenciados o auditados por agencias reguladoras. El principal objetivo de la prueba de aceptación contractual y normativa es crear confianza en que se ha logrado la conformidad contractual o normativa.
- **Prueba alfa y beta:** Las pruebas alfa y beta suelen ser utilizadas por los desarrolladores de software comercial de distribución masiva —COTS por sus siglas en inglés— que desean obtener retroalimentación de los usuarios, clientes y/u operadores potenciales o existentes antes de que el producto de software sea puesto en el mercado. La prueba alfa se realiza en las instalaciones de la organización que desarrolla, no por el equipo de desarrollo, sino por clientes potenciales o existentes, y/u operadores o un equipo de prueba independiente. La prueba beta es realizada por clientes potenciales o existentes, y/u operadores en sus propias instalaciones. La prueba beta puede tener lugar después de la prueba alfa, o puede ocurrir sin que se haya realizado ninguna prueba alfa previa.

Uno de los objetivos de las pruebas alfa y beta es generar confianza entre los clientes potenciales o existentes y/u operadores de que pueden utilizar el sistema en condiciones normales y cotidianas, así como en el entorno o los entornos operativos, para lograr sus objetivos con la mínima dificultad, coste y riesgo. Otro objetivo puede ser la detección de defectos relacionados con las condiciones y el entorno o los entornos en los que se utilizará el sistema, especialmente cuando las condiciones y los entornos sean difíciles de reproducir por parte del equipo de desarrollo.

La prueba de aceptación es, a menudo, responsabilidad de los clientes, usuarios de negocio, propietarios de producto u operadores de un sistema, y otros implicados también pueden estar involucrados. También se suele considerar como el último nivel de prueba en un ciclo de vida de desarrollo secuencial, pero pueden ocurrir en otros momentos, por ejemplo:

- La prueba de aceptación de un producto software comercial de distribución masiva —COTS por sus siglas en inglés— puede tener lugar cuando se instala o integra.
- La prueba de aceptación de una mejora funcional nueva puede tener lugar antes de la prueba de sistema.

- En el desarrollo iterativo, los equipos de proyecto pueden emplear varias formas de prueba de aceptación durante y al final de cada iteración.

		Prueba			
		Unitaria o de componente	De integración	De aceptación	De sistema
Objetivos específicos	Unitaria o de componente	+	+	+	+
	Bases de prueba	+	+	+	+
	Objeto de prueba	+	+	+	+
	Defectos y fallos característicos	+	+	+	+
	Enfoques y responsabilidades específicas	+	+	+	+

## Unitaria o de componente

### Objetivos específicos:

- Reducir el riesgo.
- Verificar que los comportamientos funcionales y no funcionales del componente son los diseñados y especificados.
- Generar confianza en la calidad del componente.
- Encontrar defectos en el componente.
- Prevenir la propagación de defectos a niveles de prueba superiores.

### Bases de prueba:

- Diseño detallado.
- Código.
- Modelo de datos.
- Especificaciones de los componentes.

### Objeto de prueba:

- Componentes, unidades o módulos.
- Código y estructuras de datos.
- Clases.
- Módulos de base de datos.
- 

### Defectos y fallos característicos:

- Funcionamiento incorrecto —por ejemplo, no lo hace de la manera en que se describe en las especificaciones de diseño—.
- Problemas de flujo de datos.

- Código y lógica incorrectos.

### **Enfoques y responsabilidades específicas:**

En general, el desarrollador que escribió el código realiza la prueba de componente. Los desarrolladores pueden alternar el desarrollo de componentes con la búsqueda y corrección de defectos. A menudo, estos escriben y ejecutan pruebas después de haber escrito el código de un componente. Sin embargo, especialmente en el desarrollo ágil, la redacción de casos de prueba de componente automatizados puede preceder a la redacción del código de la aplicación.

## **De integración**

### **Objetivos específicos:**

- Reducir el riesgo.
- Verificar que los comportamientos funcionales y no funcionales de las interfaces sean los diseñados y especificados.
- Generar confianza en la calidad de las interfaces.
- Encontrar defectos —que pueden estar en las propias interfaces o dentro de los componentes o sistemas—.
- Prevenir la propagación de defectos a niveles de prueba superiores.

### **Bases de prueba:**

- Diseño de software y sistemas.
- Diagramas de secuencia.
- Especificaciones de interfaz y protocolos de comunicación.
- Casos de uso.
- Arquitectura a nivel de componente o de sistema.
- Flujos de trabajo.
- Definiciones de interfaces externas.

### **Objeto de prueba:**

- Subsistemas.
- Bases de datos.
- Infraestructura.
- Interfaces.
- Interfaces de programación de aplicaciones —API por sus siglas en inglés—.
- Microservicios.

### **Defectos y fallos característicos:**

- Datos incorrectos, datos faltantes o codificación incorrecta de datos.
- Secuenciación o sincronización incorrecta de las llamadas a la interfaz.
- Incompatibilidad de la interfaz.
- Fallos en la comunicación entre componentes.
- Fallos de comunicación entre componentes no tratados o tratados de forma incorrecta.
- Suposiciones incorrectas sobre el significado, las unidades o las fronteras de los datos que se transmiten entre componentes.

### **Enfoques y responsabilidades específicas:**

La prueba de integración debe concentrarse en la integración propiamente dicha. Se puede utilizar los tipos de prueba funcional, no funcional y estructural. En general es responsabilidad de los testers.

## De aceptación

### Objetivos específicos:

La prueba de aceptación, al igual que la prueba de sistema, se centra normalmente en el comportamiento y las capacidades de todo un sistema o producto. Los objetivos de la prueba de aceptación incluyen:

- Establecer confianza en la calidad del sistema en su conjunto.
- Validar que el sistema está completo y que funcionará como se espera.
- Verificar que los comportamientos funcionales y no funcionales del sistema sean los especificados.

### Bases de prueba:

- Procesos de negocio.
- Requisitos de usuario o de negocio.
- Normativas, contratos legales y estándares.
- Casos de uso.
- Requisitos de sistema.
- Documentación del sistema o del usuario.
- Procedimientos de instalación.
- Informes de análisis de riesgo.

### Objeto de prueba:

- Sistema sujeto a prueba.
- Configuración del sistema y datos de configuración.
- Procesos de negocio para un sistema totalmente integrado.
- Sistemas de recuperación y sitios críticos —para pruebas de continuidad del negocio y recuperación de desastres—.
- Procesos operativos y de mantenimiento.
- Formularios.
- Informes.
- Datos de producción existentes y transformados.

### Defectos y fallos característicos:

- Los flujos de trabajo del sistema no cumplen con los requisitos de negocio o de usuario.
- Las reglas de negocio no se implementan de forma correcta.
- El sistema no satisface los requisitos contractuales o reglamentarios.
- Fallos no funcionales tales como vulnerabilidades de seguridad, eficiencia de rendimiento inadecuada bajo cargas elevadas o funcionamiento inadecuado en una plataforma soportada.

### Enfoques y responsabilidades específicas

A menudo es responsabilidad de los clientes, usuarios de negocio, propietarios de producto u operadores de un sistema, y otros implicados también pueden estar involucrados. La prueba de aceptación se considera, a menudo, como el último nivel de prueba en un ciclo de vida de desarrollo secuencial.

## De sistema

### Objetivos específicos:

- Reducir el riesgo.
- Verificar que los comportamientos funcionales y no funcionales del componente son los diseñados y especificados.
- Validar que el sistema está completo y que funcionará como se espera.
- Generar confianza en la calidad del sistema considerado como un todo.

- Encontrar defectos.
- Prevenir la propagación de defectos a niveles de prueba superiores o a producción.

### **Bases de prueba:**

- Especificaciones de requisitos del sistema y del software —funcionales y no funcionales—.
- Informes de análisis de riesgo.
- Casos de uso.
- Épicas e historias de usuario.
- Modelos de comportamiento del sistema.
- Diagramas de estado.
- Manuales del sistema y del usuario.

### **Objeto de prueba:**

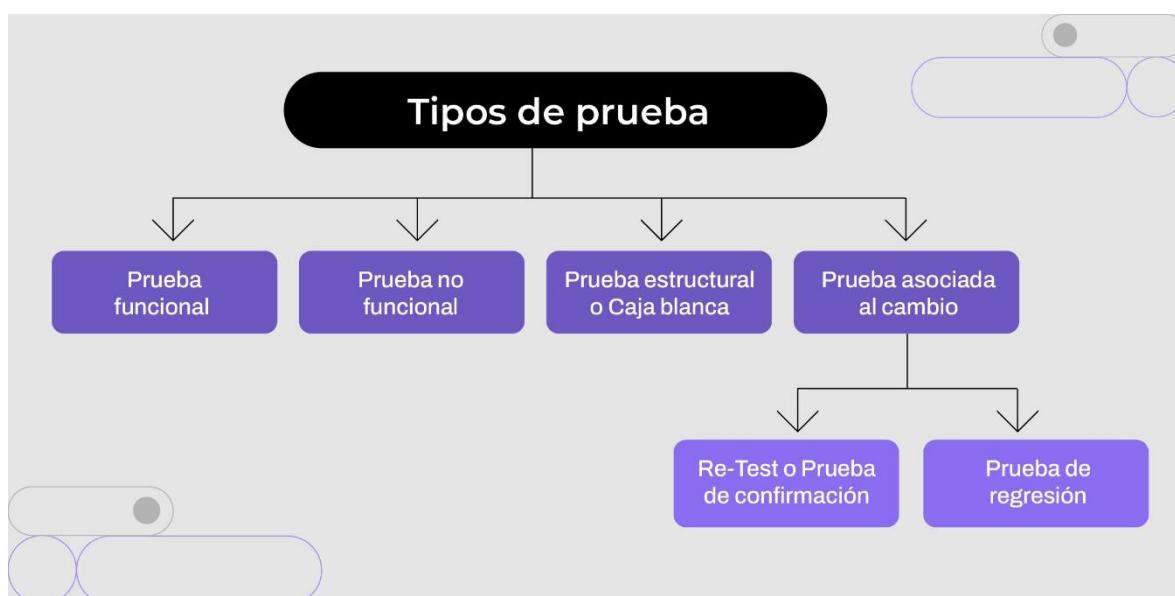
- Aplicaciones.
- Sistemas hardware/software.
- Sistemas operativos.
- Sistema sujeto a prueba (SSP).
- Configuración del sistema y datos de configuración.

### **Defectos y fallos característicos:**

- Cálculos incorrectos.
- Comportamiento funcional o no funcional del sistema incorrecto o inesperado.
- Control y/o flujos de datos incorrectos dentro del sistema.
- Incapacidad para llevar a cabo, de forma adecuada y completa, las tareas funcionales extremo a extremo.
- Fallo del sistema para operar correctamente en el/los entorno/s de producción.
- Fallo del sistema para funcionar como se describe en los manuales del sistema y de usuario.

### **Enfoques y responsabilidades específicas:**

La prueba de sistema debe centrarse en el comportamiento global y extremo a extremo del sistema en su conjunto, tanto funcional como no funcional. Debe utilizar las técnicas más apropiadas para los aspectos del sistema que serán probados. Los probadores independientes, en general, llevan a cabo la prueba de sistema.



# Tipos de prueba

Un tipo de prueba es un grupo de actividades de pruebas destinadas a probar las características específicas de un sistema de software, o de una parte de un sistema, basados en objetivos de pruebas específicas.

Dichos objetivos pueden incluir:

1. Evaluar las características de calidad funcional tales como la completitud, corrección y pertinencia.
2. Evaluar características no funcionales de calidad, tales como la fiabilidad, eficiencia de desempeño, seguridad, confiabilidad y usabilidad.
3. Evaluar si la estructura o arquitectura del componente o sistema es correcta, completa y según lo especificado.
4. Evaluar los efectos de los cambios, tales como confirmar que los defectos han sido corregidos — prueba de confirmación — y buscar cambios no deseados en el comportamiento que resulten de los cambios en el software o en el entorno — prueba de regresión —.

	1. Prueba Funcional	2. Prueba no funcional	3. Prueba estructural	4. Prueba asociada al cambio
<b>Definición</b>	La prueba funcional de un sistema incluye pruebas que evalúan las funciones que el sistema debe realizar. Las funciones describen <b>qué hace</b> el sistema.	La prueba no funcional prueba <b>“cómo de bien”</b> se comporta el sistema.	Estas pruebas están basadas en la estructura interna del sistema o en su implementación. La estructura interna puede incluir código, arquitectura, flujos de trabajo y/o flujos de datos dentro del sistema	Existen 2 tipos de prueba relacionadas al cambio: <ul style="list-style-type: none"><li>• <b>Prueba de confirmación:</b> una vez corregido un defecto, el software se puede probar con todos los casos de prueba que fallaron debido al defecto, que se deben volver a ejecutar en la nueva versión de software. El objetivo de una prueba de confirmación es confirmar que el defecto original se ha</li></ul>

	<b>1. Prueba Funcional</b>	<b>2. Prueba no funcional</b>	<b>3. Prueba estructural</b>	<b>4. Prueba asociada al cambio</b>
				<p>solucionado de forma satisfactoria.</p> <ul style="list-style-type: none"> <li>• <b>Prueba de regresión:</b> es posible que un cambio hecho en una parte del código, ya sea una corrección u otro tipo de cambio, pueda afectar accidentalmente el comportamiento de otras partes del código, ya sea dentro del mismo componente, en otros componentes del mismo sistema, o incluso en otros sistemas. La prueba de regresión implica la realización de pruebas para detectar estos efectos secundarios no deseados.</li> </ul>
<b>Implementación</b>	La prueba funcional observa el	El diseño y ejecución de la prueba no	El diseño y la ejecución de este tipo de pruebas pueden implicar	Especialmente en los ciclos de vida de desarrollo iterativos e

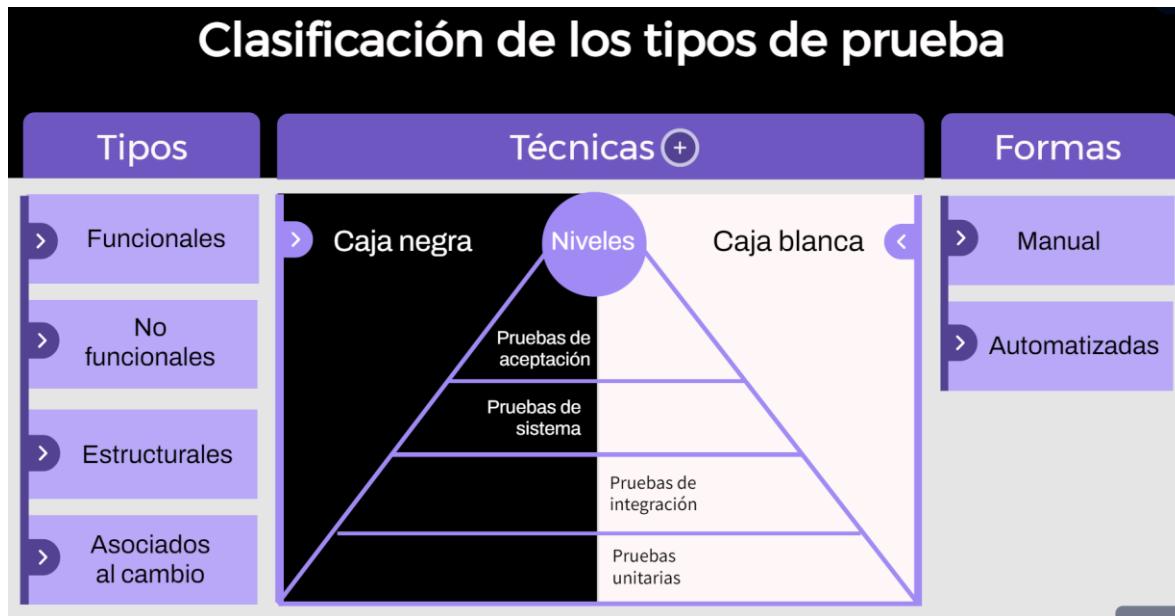
	<b>1. Prueba Funcional</b>	<b>2. Prueba no funcional</b>	<b>3. Prueba estructural</b>	<b>4. Prueba asociada al cambio</b>
	comportamiento del software.	funcional puede implicar competencias y conocimientos especiales, como el conocimiento de las debilidades inherentes a un diseño o tecnología — por ejemplo: vulnerabilidades de seguridad asociadas con determinados lenguajes de programación —.	competencias o conocimientos especiales, como la forma en que se construye el código, cómo se almacenan los datos, y cómo utilizar las herramientas de cobertura e interpretar correctamente sus resultados.	incrementales —por ejemplo, Agile—, las nuevas características, los cambios en las características existentes y la refactorización del código dan como resultado cambios frecuentes en el código, lo que también requiere pruebas asociadas al cambio.
<b>Niveles de prueba</b>	Se pueden realizar pruebas funcionales en todos los niveles de prueba.	Se pueden realizar pruebas no funcionales en todos los niveles de prueba	Se puede realizar en el nivel de componente y de integración.	La prueba de confirmación y la prueba de regresión se realizan en todos los niveles de prueba.
<b>Alcance</b>	Los requisitos funcionales pueden estar detallados en los siguientes documentos: especificaciones de requisitos del negocio, épicas, historias de usuarios, casos de uso y/o especificaciones funcionales.	La prueba no funcional del sistema evalúa características como la usabilidad, la eficiencia del desempeño o la seguridad.	En el nivel de prueba de integración de componentes, la prueba estructural puede basarse en la arquitectura del sistema, como las interfaces entre componentes.	

	<b>1. Prueba Funcional</b>	<b>2. Prueba no funcional</b>	<b>3. Prueba estructural</b>	<b>4. Prueba asociada al cambio</b>
<b>Cobertura</b>	La cobertura funcional es la medida en que algún tipo de elemento funcional ha sido practicado por pruebas, y se expresa como un porcentaje del tipo o tipos de elementos cubiertos.	La cobertura no funcional es la medida en que algún tipo de elemento no funcional ha sido practicado por pruebas, y se expresa como un porcentaje del tipo o tipos de elementos cubiertos.	La cobertura estructural es la medida en que algún tipo de elemento estructural ha sido practicado mediante pruebas, y se expresa como un porcentaje del tipo de elemento cubierto.	Los juegos de prueba de regresión se ejecutan muchas veces y generalmente evolucionan lentamente, por lo que la prueba de regresión es un fuerte candidato para la automatización. La cobertura crece a medida que se agregan más funcionalidades al sistema por lo tanto más pruebas de regresión.

### **Ejemplos de pruebas no funcionales:**

- **Pruebas de recuperación:** para verificar qué tan rápido y qué tan bien se recupera una aplicación luego de experimentar un fallo de hardware o software.
- **Pruebas de mantenibilidad:** consisten en evaluar qué tan fácil es realizar el mantenimiento de una aplicación o sistema. Es decir, qué tan fácil es analizar, cambiar y probar estos cambios.
- **Pruebas de seguridad:** consisten en probar atributos o características de seguridad del sistema, si es un sistema seguro o no, si puede ser vulnerado, si existe control de acceso por medio de cuentas de usuario, si pueden ser vulnerados esos accesos, etc.
- **Pruebas de resistencia:** implican someter a una aplicación a una carga determinada durante un período de tiempo para determinar cómo se comporta luego de un uso prolongado.
- **Pruebas de volumen:** consisten en validar el funcionamiento de la aplicación con ciertos volúmenes de datos.

- **Pruebas de estrés:** son pruebas de carga que se realizan con demandas mayores a la capacidad operativa, con frecuencia hasta llegar al punto de ruptura.
- **Pruebas de carga:** consisten en simular la demanda sobre una aplicación de software y medir el resultado y uso de recursos. Estas pruebas se realizan bajo demanda esperada y también en condiciones de sobrecarga o picos en la demanda.



## Error, defecto y falla

Debemos tener claro el concepto de error, defecto y falla para poder utilizarlos correctamente. Como se confunden fácilmente entre sí, vamos a comprobar en detalle la definición de cada uno.

El **error** es lo que una persona comete durante el ciclo de desarrollo del software, como una línea de código o una definición incorrecta de un requisito. Un error causará un defecto en el software.

El **defecto** caracteriza por un resultado incorrecto o inesperado, derivado de un error en el software. Por lo general, lo encuentra el equipo de pruebas.

Una **falla** es un resultado inesperado tras la ejecución —por parte del usuario final— de un defecto.

Es importante tener en cuenta:

- Algunos defectos requieren entradas o condiciones previas muy específicas para desencadenar un fallo, que puede ocurrir raramente o nunca.
- Son llamados falsos negativos, aquellas pruebas que no detectan defectos que deberían ser detectados.
- Son llamados falsos positivos aquellas que se notifican como defectos pero que en realidad no lo son.
- Identificar un defecto es muy importante para que el software funcione correctamente cuando el usuario vaya a utilizarlo. Por eso es tan importante la función de un equipo de calidad. Cuanto antes —dentro del proceso de desarrollo de software— se encuentre el defecto, más barato será su coste —de corrección, impacto—.

## Reporte de defectos

Cuando se detecta un defecto, es crucial que se corrija adecuadamente, por lo que es necesario llenar lo que llamamos un **informe de defectos**.

El defecto debe tener algunas características para ser registrado: **ser reproducible y ser específico**. Además, es fundamental que se informe todo el paso a paso para reproducir el defecto, para ayudar al desarrollador en el proceso de corrección y optimizar el tiempo de análisis. Se debe asignar un identificador único para facilitar la identificación y el seguimiento.

En general, **los campos importantes de un informe de defectos son:**

- 1- Un identificador único
- 2- Un título
- 3- Breve descripción del defecto
- 4- Fecha del informe
- 5- Autor
- 6- Identificación del componente de prueba
- 7- Versión de la aplicación
- 8- Ambiente
- 9- Pasos para su reproducción
- 10- Resultado esperado
- 11- Resultado obtenido
- 12- Severidad

13- Prioridad

14- Estado del defecto

15- Imagen del defecto

Además, hay algunas **buenas prácticas** para evitar los problemas de los informes:

- Incluir en la descripción el resultado esperado.
- Ser directo con el problema.
- Releer y validar que la descripción es clara, sin ambigüedades y no demasiado coloquial.
- Incluir toda la información pertinente relacionada con las características del defecto.
- La gravedad y la prioridad pueden ser informadas para indicar el impacto y ayudar a priorizar la corrección.

## Partes de un informe de defectos

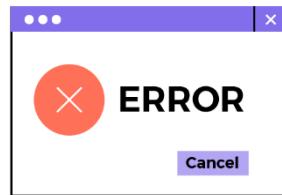
Atributo	Descripción	Ejemplo
ID	Abreviatura de identificador, un código único e irrepetible que puede ser número o letras.	001 - Test01
Título	El título debe ser corto y específico, que se entienda en este lo que queremos reportar. Cuando el desarrollador o el equipo vean el título pueden interpretar rápidamente qué es, dónde está y cuán importante es ese defecto.	Login - Ingresa con campos en blanco
Descripción	Describir un poco más sobre el error, es decir, desarrollar lo que dejamos afuera en el título y que lo podríamos explicar acá.	En la pantalla login si dejo vacío los campos nombre y password y aprieto ingresar, me lleva a la página principal.
Fecha del informe del defecto	La fecha que detectó el defecto para saber posteriormente el tiempo en que se resolvió.	23/04/21
Autor	El nombre del tester que descubrió el defecto, por si el desarrollador tiene una duda, sabe a quién consultar.	Pepito Román
Identificación del elemento de prueba	Nombre de la aplicación o componente que estamos probando.	Carrito compras

Atributo	Descripción	Ejemplo
<b>Versión</b>	Es un número que nos indica en qué versión está la aplicación.	1.0.0
<b>Entorno</b>	El entorno en el que probamos —desarrollo, QA, producción—.	Desarrollo
<b>Pasos a reproducir</b>	Los pasos a seguir para llegar al defecto encontrado.	1) Ingresar a la aplicación. 2) Dejar en blanco el campo nombre. 3) Dejar en blanco el campo password. 4) Hacer clic en el botón “Ingresar”.
<b>Resultado esperado</b>	Es lo que esperamos que suceda o muestre la aplicación muchas veces según los requerimientos de la misma.	No debe ingresar a la aplicación sin un usuario y una contraseña válidos.
<b>Resultado obtenido o actual</b>	Es lo que sucedió realmente o lo que nos mostró la aplicación. Puede coincidir o no con el resultado esperado, si no coincide, hemos detectado un error o bug.	Ingresar a la aplicación sin usuario y sin contraseña.
<b>Severidad</b>	Cuán grave es el defecto que hemos encontrado, puede ser: bloqueado, crítico, alto, medio, bajo o trivial.	Crítico
<b>Prioridad</b>	Con esto decimos qué tan rápido se debe solucionar el defecto, puede ser: alta, media, baja.	Alta
<b>Estado del defecto</b>	Los estados pueden ser: nuevo, diferido, duplicado, rechazado, asignado, en progreso, corregido, en espera de verificación, en verificación, verificado, re-abierto y cerrado.	Nuevo

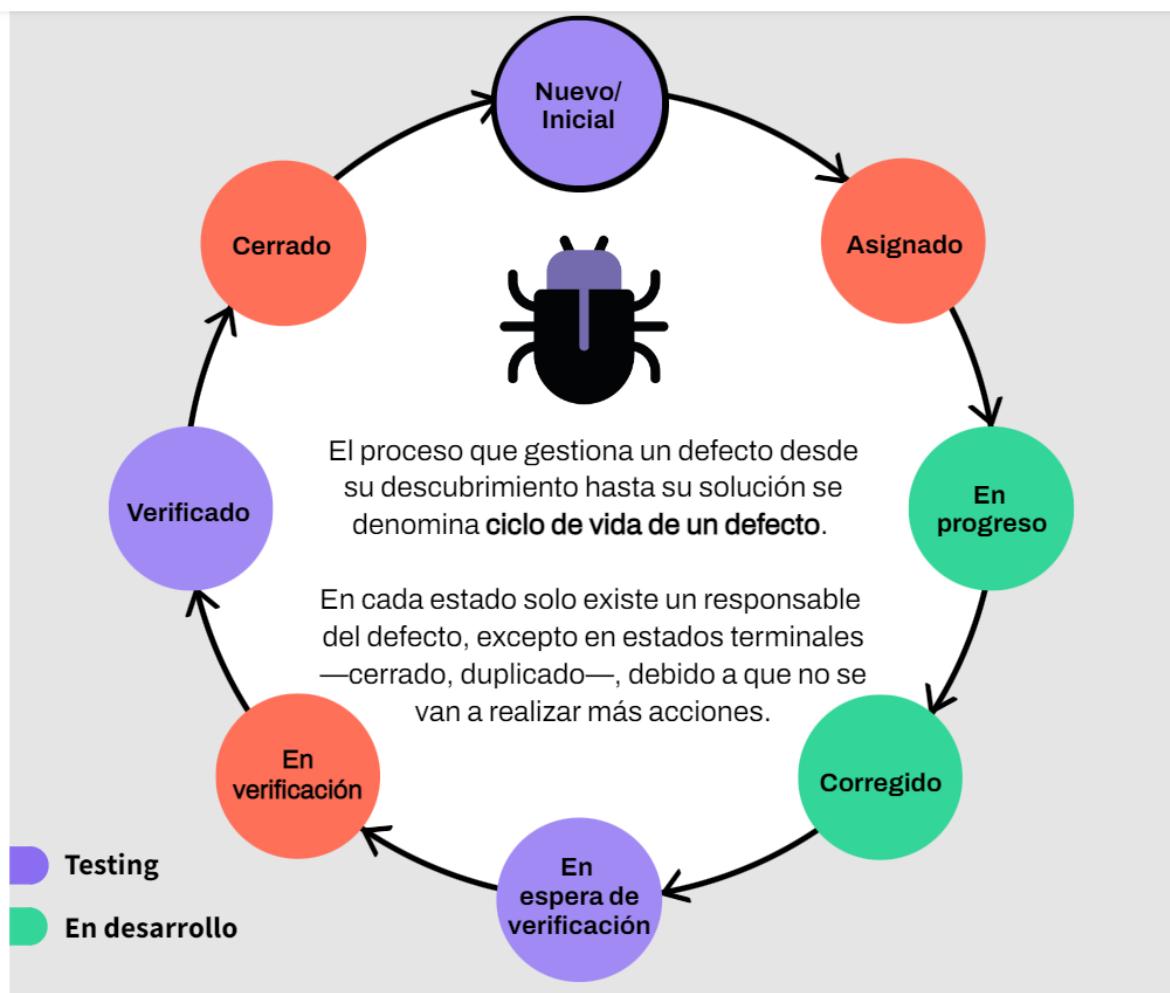
Atributo	Descripción	Ejemplo
<b>Referencias</b>	Link al caso de prueba con el cual encontramos el error.	<a href="https://repositorio.com.ar/TC-001-UserLogin">https://repositorio.com.ar/TC-001-UserLogin</a>

## Imagen

Se puede adjuntar una captura de pantalla del error, esto nos permite demostrar que el error sucedió y al desarrollador lo ayuda a ubicar el error



## Ciclo de vida de un defecto



Por lo general, al principio del desarrollo de software pueden producirse **errores**. Cuando se prueba el software, se encuentran **defectos**, defectos que no se identifiquen antes de que el software se libere para su uso o pase a producción provocarán **fallas**.

**Los defectos deben registrarse en informes** para que la información esencial se transmita al equipo de desarrollo para ayudar en el proceso de corrección. No existe un modelo estándar, pero hemos revisado las mejores prácticas y las principales características del defecto que se puede notificar. Cuando se informa de un defecto de forma eficiente —especialmente describiendo el paso a paso—, hay muchas posibilidades de que la corrección se realice en menos tiempo.

**Existe un ciclo de vida del defecto**, con el objetivo principal de hacer un seguimiento del mismo, ya que marca el inicio y el final de la existencia de un determinado defecto. Hay estados asignados en función de la fase del proceso de corrección en la que se encuentre. Un ciclo común pasa por los siguientes **estados: NUEVO, ADJUNTO, EN PROCESO, ARREGLADO, EN ESPERA DE VERIFICACIÓN, EN VERIFICACIÓN, VERIFICADO y CERRADO**.

## ¿Por qué automatizar tests?

Las pruebas de software deben repetirse con frecuencia durante los ciclos de desarrollo para garantizar la calidad. Cada vez que se modifica el código fuente, hay que repetir las pruebas de software.

Repetir manualmente estas pruebas es costoso y lleva mucho tiempo. Una vez creadas, las pruebas automatizadas pueden ejecutarse repetidamente sin coste adicional y son mucho más rápidas que las pruebas manuales.

La automatización de las pruebas puede ejecutar fácilmente miles de casos de prueba complejos diferentes durante cada ejecución de la prueba, proporcionando una cobertura que es imposible con las pruebas manuales.

En resumen, las pruebas de software automatizadas pueden reducir el tiempo de ejecución de las pruebas repetitivas y, en consecuencia, podemos ahorrar costos.

## ¿Qué es Selenium Webdriver?

Selenium Webdriver es uno de los frameworks más utilizados en el mercado. Se trata de una colección de APIs de código abierto que se utiliza para automatizar las pruebas de aplicaciones y sistemas web. Es compatible con los principales navegadores como Firefox, Chrome, Safari e Internet Explorer, y también permite el uso de varios lenguajes de programación (Java, .Net, PHP, Python, Perl, Ruby) en la creación de sus scripts de prueba.

Conozcamos ahora cuáles son sus principales ventajas y desventajas.

 Ventajas	 Desventajas
Solución de código abierto con actualizaciones constantes. Al ser una herramienta de código abierto, Selenium es un marco de automatización de acceso público sin costes iniciales.	Selenium WebDriver no es capaz de manejar componentes con ventanas.
Selenium WebDriver permite elegir un lenguaje de programación para crear scripts de prueba.	No tiene ninguna capacidad de reporte incorporada, necesita depender de plugins como JUnit y TestNG para los reportes de pruebas.
Selenium WebDriver completa la ejecución del script de prueba más rápido en comparación con otras herramientas.	No es posible realizar pruebas en imágenes.
Selenium se integra fácilmente con varias plataformas de desarrollo como Jenkins, Maven, TestNG, QMetry, SauceLabs, etc.	No hay capacidades de reporte, uno de los mayores desafíos de Selenium. Para capturar los fallos de las pruebas, hay que hacer una captura de pantalla en el momento del fallo.
Es compatible con las pruebas móviles, aunque requiere un software adicional. Hay dos opciones principales: Appium y Selendroid.	

## Problemas comunes al automatizar

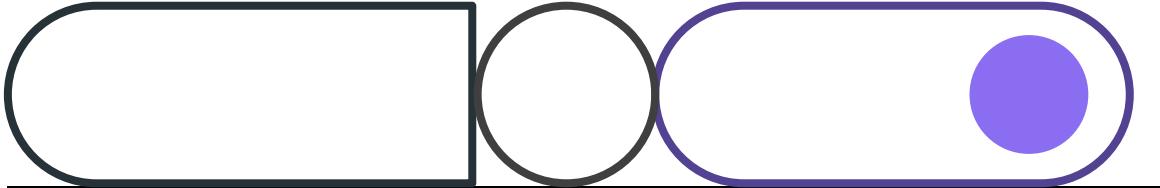
La automatización no siempre es una buena idea si las prácticas de prueba son deficientes, con pruebas mal organizadas, documentación incoherente y pruebas que no son muy buenas para encontrar defectos. Lo primero y más importante es mejorar la eficacia de las pruebas.

El hecho de que las suites de prueba no encuentren ningún defecto, no significa que no haya ningún defecto en el software. Esto es crucial porque si las pruebas contienen defectos, darán resultados incorrectos. Las pruebas de automatización simplemente conservarán esos resultados defectuosos indefinidamente.

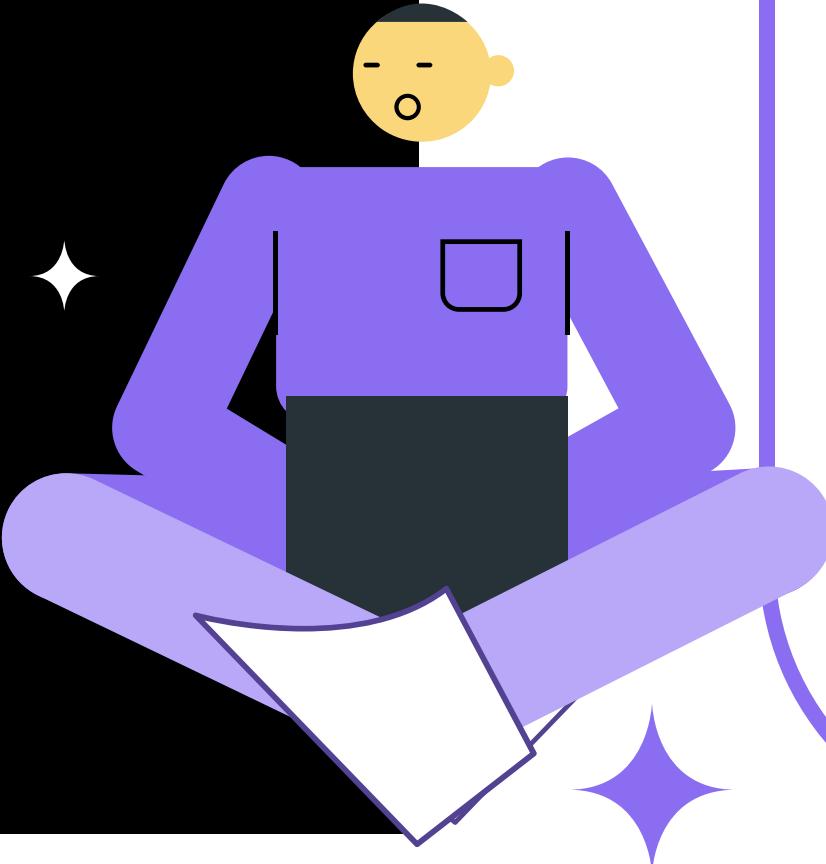
Cada vez que se modifica o actualiza el software, también deben actualizarse sus pruebas para que puedan volver a ejecutarse con éxito. El mantenimiento de las pruebas ha sido la muerte de muchas iniciativas de automatización de pruebas. Cuando la actualización de las pruebas requiera más esfuerzo que el que supondría ejecutarlas manualmente, se abandonará la automatización de las pruebas. Su iniciativa de automatización de pruebas no debe ser víctima de los altos costes de mantenimiento.

01

¿Qué es un patrón?



Una de las definiciones más destacada es la siguiente:  
“Los patrones de diseño son el esqueleto de las soluciones a problemas comunes en el desarrollo de software.”



# Patrones

Brindan una solución ya **probada** y **documentada** a problemas de desarrollo de software que están sujetos a contextos similares. Los siguientes elementos se deben tener en cuenta a la hora de elegir e implementar un patrón:

- Su nombre
- El problema: cuándo aplicar un patrón
- La solución: descripción abstracta del problema
- Las consecuencias: costos y beneficios

# ¿Dónde surgieron?

Antes de comenzar a detallar cada patrón deberíamos saber de dónde surgen los patrones y cuál es su utilidad.

El concepto de patrón de diseño lleva existiendo desde **finales de los 70**, pero su verdadera popularización surgió en los 90 con el **lanzamiento del libro de Design Pattern** de la Banda de los Cuatro (Gang of Four), nombre con el que se conoce a Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides. En este libro explican 23 patrones de diseño, que desde entonces han sido considerados una referencia.

# ¿Por qué son útiles los patrones de diseño?

## Te ahorran tiempo

Buscar siempre una nueva solución a los mismos problemas reduce tu eficacia como desarrollador. No hay que olvidar que **el desarrollo de software también es una ingeniería**, y que por tanto en muchas ocasiones habrá reglas comunes para solucionar problemas comunes.

## Te ayudan a estar seguro de la validez de tu código

Los patrones de diseño son estructuras probadas por millones de desarrolladores a lo largo de muchos años, por lo que si eligen el patrón adecuado para modelar el problema adecuado, pueden estar seguros de que va a ser **una de las soluciones más válidas**.

## Establecen un lenguaje común

Modelar sus códigos mediante patrones les ayudará a explicar a otras personas, conozcan su código o no, a entender cómo han solucionado un problema. **Ayudan a todo el equipo a avanzar mucho más rápido**, con un código más fácil de entender y mucho más robusto.

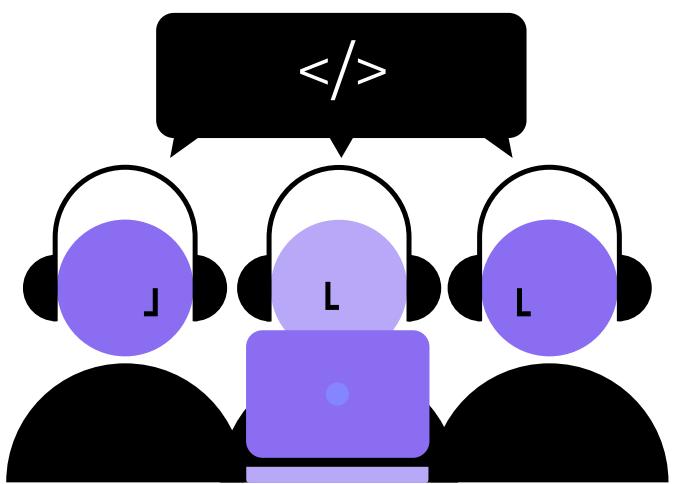
02

## Principios SOLID

# Principios SOLID

Los patrones de diseño surgen debido a la aplicación de buenas prácticas, que están regidas por los principios SOLID de la programación Orientada a Objetos. Entonces: ¿qué son los principios SOLID?

“Son un conjunto de **principios aplicables a la Programación Orientada a Objetos** que, si los usas correctamente, **te ayudarán a escribir software de calidad** en cualquier lenguaje de programación orientada a objetos. Gracias a ellos, crearás código que será más fácil de leer, testear y mantener.”

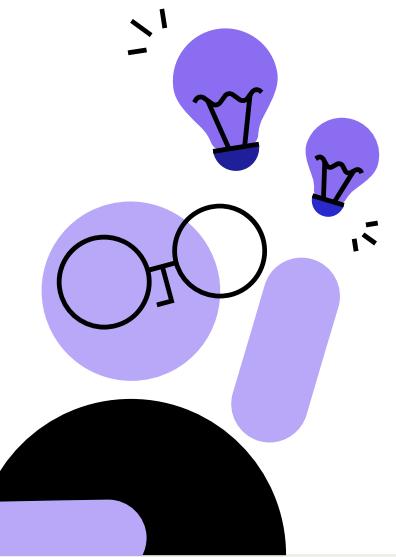


# Principios SOLID

Los principios en los que se basa **SOLID** son los siguientes:

- Principio de Responsabilidad Única (**S**ingle Responsibility Principle)
- Principio Open/Closed (**O**pen/Closed Principle)
- Principio de Sustitución de Liskov (**L**iskov Substitution Principle)
- Principio de Segregación de Interfaces (**I**nterface Segregation Principle)
- Principio de Inversión de Dependencias (**D**evelopment Dependency Inversion Principle)

Fueron publicados por primera vez por Robert C. Martin, también conocido como Uncle Bob, en su libro Agile Software Development: Principles, Patterns, and Practices.



# ¿Qué beneficios aportan los Principios SOLID?

Las **ventajas** de utilizar los **Principios SOLID** son innumerables, ya que nos aportan todas esas características que siempre queremos ver en un software de calidad.

- Software más flexible: mejoran la cohesión disminuyendo el acoplamiento: a grandes rasgos lo que buscamos de un buen código es que sus clases puedan trabajar de forma independiente y que el cambio de uno afecte lo menos posible al resto.
- Les van a hacer entender mucho mejor las arquitecturas.
- Simplifican la creación de tests.

Al final piensa que todo es como una cadena: si se aplican bien los principios, se organiza mejor el código. Esto permite definir una arquitectura que hará que los tests sean más sencillos.

Sabiendo de dónde surgen los patrones y cuáles son sus beneficios, ahora sí, vamos a detallar los patrones utilizados en el diseño de código correspondiente a pruebas automatizadas desde la perspectiva de los elementos que lo componen.

03

# Patrón Page Object

# Patrón Page Object

Este es el patrón de diseño más implementado para mejorar el mantenimiento de las pruebas y reducir el código duplicado. El concepto detrás de este patrón es el de **representar cada una de las pantallas que componen el sitio web o la aplicación que nos interesa probar**, como una serie de objetos que encapsulan las características y funcionalidades representadas en la página.

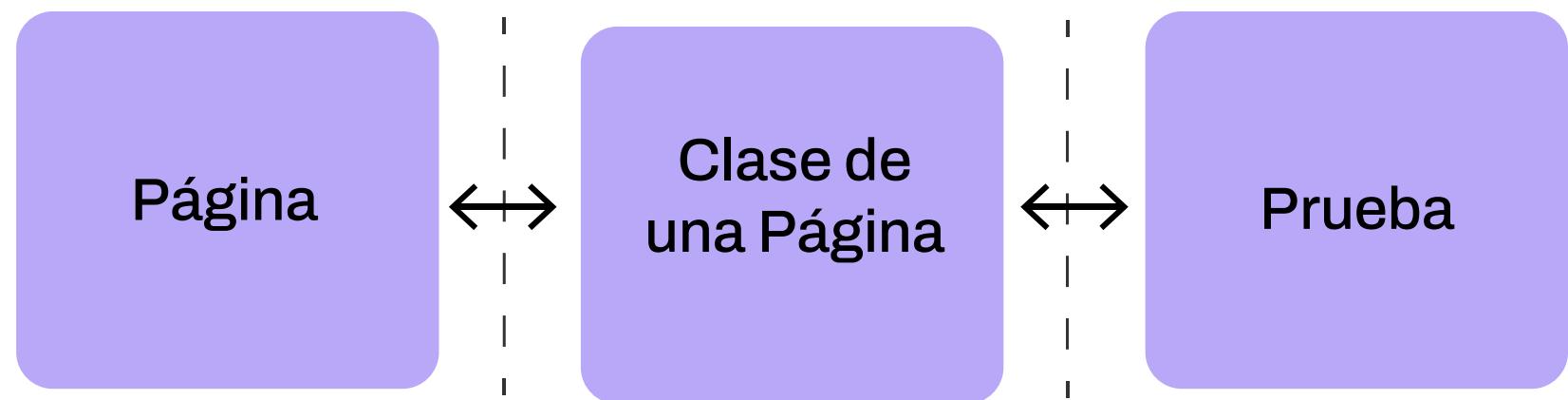
Al implementar este patrón, lo que estamos consiguiendo es crear una capa de abstracción entre el “¿qué podemos hacer/ver en la página?”, y el “¿cómo se realiza esta acción?”, simplificando enormemente la creación de nuestras pruebas y reutilizando el código con el que interactuamos con la página en concreto.

# Patrón Page Object

Al implementar PageObject agrupamos aquellos métodos comunes a todas las páginas y luego vamos creando un PageObject para cada grupo de elementos que se relacionen de alguna manera —ya sea una barra de menú que se repite en varias páginas, una página con una funcionalidad específica, un popup que aparece en varias páginas, etc.—. Lo mismo va a aplicar para los tests, ya que podemos centralizar en un TestPage object todo lo relacionado a la configuración de los tests y métodos de validación/generación de datos.

Por lo tanto, cualquier cambio que se produzca en la UI únicamente afectará al PageObject en cuestión, no a los test ya implementados.

De esta manera, el PageObject se convierte en una API con la que fácilmente podemos encontrar y manipular los datos de la página.



Se puede dar el caso en el que una página corresponda a más de un PageObject, cuando algunas áreas de la página son lo suficientemente significativas. Por ejemplo, en una página web, podemos tener un PageObject para el header y otro para el body.

# Patrón Page Object

Una opción válida al momento de implementación es crear, en primer lugar, una clase “básica”, que posteriormente será la que extenderán cada uno de los distintos PageObjects que vayamos implementando. Esta PageBase nos aporta la estructura básica y las propiedades generales que utilizaremos:



# Patrón Page Object

Y una vez creada esta página, crear los PageObjects necesarios.

Un ejemplo de una página de “Login” podría ser:



03

# Patrón Screenplay

# Patrón Screenplay

**Screenplay** surge al escribir nuestras pruebas automatizadas basadas en los principios de ingeniería de software, SOLID, tal como son el Principio de Responsabilidad Única (SRP) y en el Principio Open/Closed (OCP).

El patrón Screenplay ha estado desde el 2007, algunos de los precursores son Antonio Marcano, Andy Palmer, Jan Molak y Jason Ferguson.

**Screenplay promueve los buenos hábitos de testing** y suites de pruebas bien organizadas las cuales son fáciles de seguir, fáciles de mantener y fáciles de extender; permitiendo que los equipos escriban pruebas automatizadas más robustas y entendibles.

Con este patrón las pruebas describen como un usuario puede interactuar con la aplicación para alcanzar una meta, es decir, está orientado al comportamiento del usuario, al enfoque de desarrollo encaminado por comportamiento o Behaviour Driven Development (BDD).

# Patrón Screenplay

Un usuario interactuando con el sistema es llamado "Actor". **Los Actores son la parte central del patrón Screenplay.** Como Screenplay (Guion) los actores tienen uno o más "habilidades", como son la habilidad de navegar por la web o realizar una consulta un web service, el "Actor" también puede realizar "Tareas", como pueden ser agregar o remover objetos o cosas.

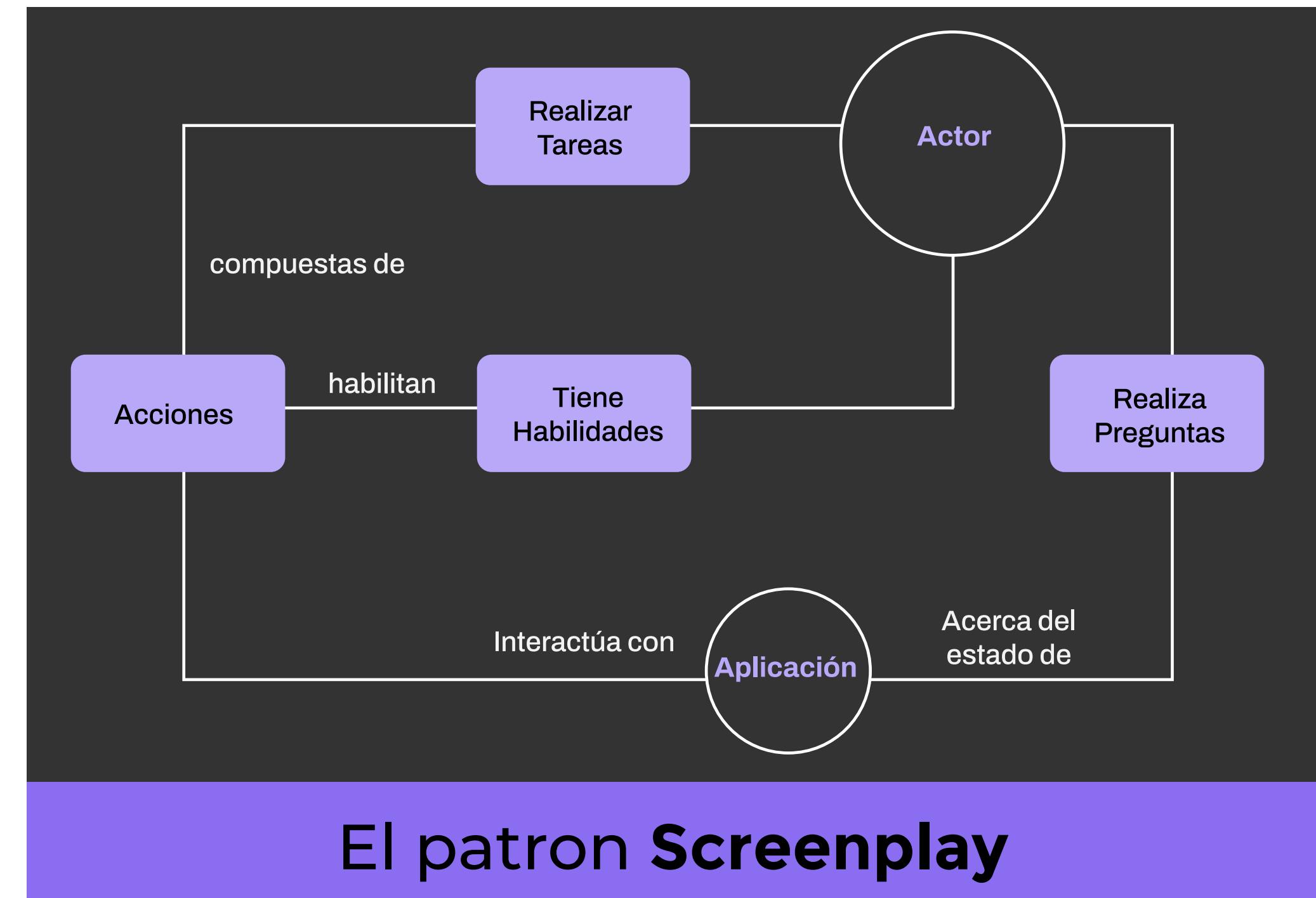
Para interactuar con la aplicaciones, como pueden ser introducir valores en los campos o dar clic en botones, los actores necesitan estas interacciones, que son denominadas "Acciones". Los "Actores" pueden hacer "Preguntas" acerca del estado del sistema, como por ejemplo, leer un valor de un campo en la pantalla o realizando una consulta a algún web service.



# Patrón Screenplay

Es decir, el enfoque está basado en los siguientes elementos:

- Actores
- Habilidades
- Tareas
- Acciones
- Preguntas



04

# Behavior Driven Development (BDD)

# ¿Qué es BDD (Behavior Driven Development)?

Este se refiere a desarrollo dirigido por comportamientos, BDD se trata de una estrategia de desarrollo, que se enfoca en prevenir defectos en lugar de encontrarlos en un ambiente controlado.

BDD es una **estrategia muy utilizada en las metodologías ágiles**, ya que en estas generalmente los requerimientos son entregados como User Stories (Historias de usuario). Lo que se plantea a través del desarrollo dirigido por comportamientos es la definición de un lenguaje común para el negocio y equipo solucionador.

Con BDD las pruebas de aceptación podrían ser escritas directamente en lenguaje Gherkin sin ningún problema, ya que este es un lenguaje común que puede escribir alguien sin conocimientos en programación.

Un ejemplo para el Login de una aplicación en lenguaje Gherkin:

## Feature File:

```
1 Feature: login
2 Scenario: login with valid credentials
3   Given I am on the login page
4   When I enter a valid email
5   And I enter a valid password
6   And I press "Login"
7   Then I should be on the users home page
8   And I should see "Login successful"
```

# BDD

El lenguaje **Gherkin** no es más que un texto con una estructura de 5 palabras claves con las que construimos sentencias y con estas describimos las funcionalidades. Este texto se guarda en un archivo con la extensión “.feature”, las palabras claves que normalmente se utilizan son:

- **Feature:** nombre de la funcionalidad que vamos a probar, el título de la prueba (HU).
- **Scenario:** por cada prueba que se ejecutara se especifica la funcionalidad a probar.
- **Given:** este sería el contexto del escenario de prueba o las precondiciones de los datos.
- **When:** se especifican el conjunto de las acciones que se van a ejecutar en el test.
- **Then:** aquí se especifica el resultado esperado del test y/o las validaciones que deseamos realizar.
-

# BDD

Las herramientas como **Cucumber**, **JBehave**, **Specflow** y muchas otras, permiten la implementación de una capa de conexión entre lenguaje Gherkin y la interfaz de usuario que se desea probar, pudiendo así utilizar eso como los pasos de una prueba automatizada.

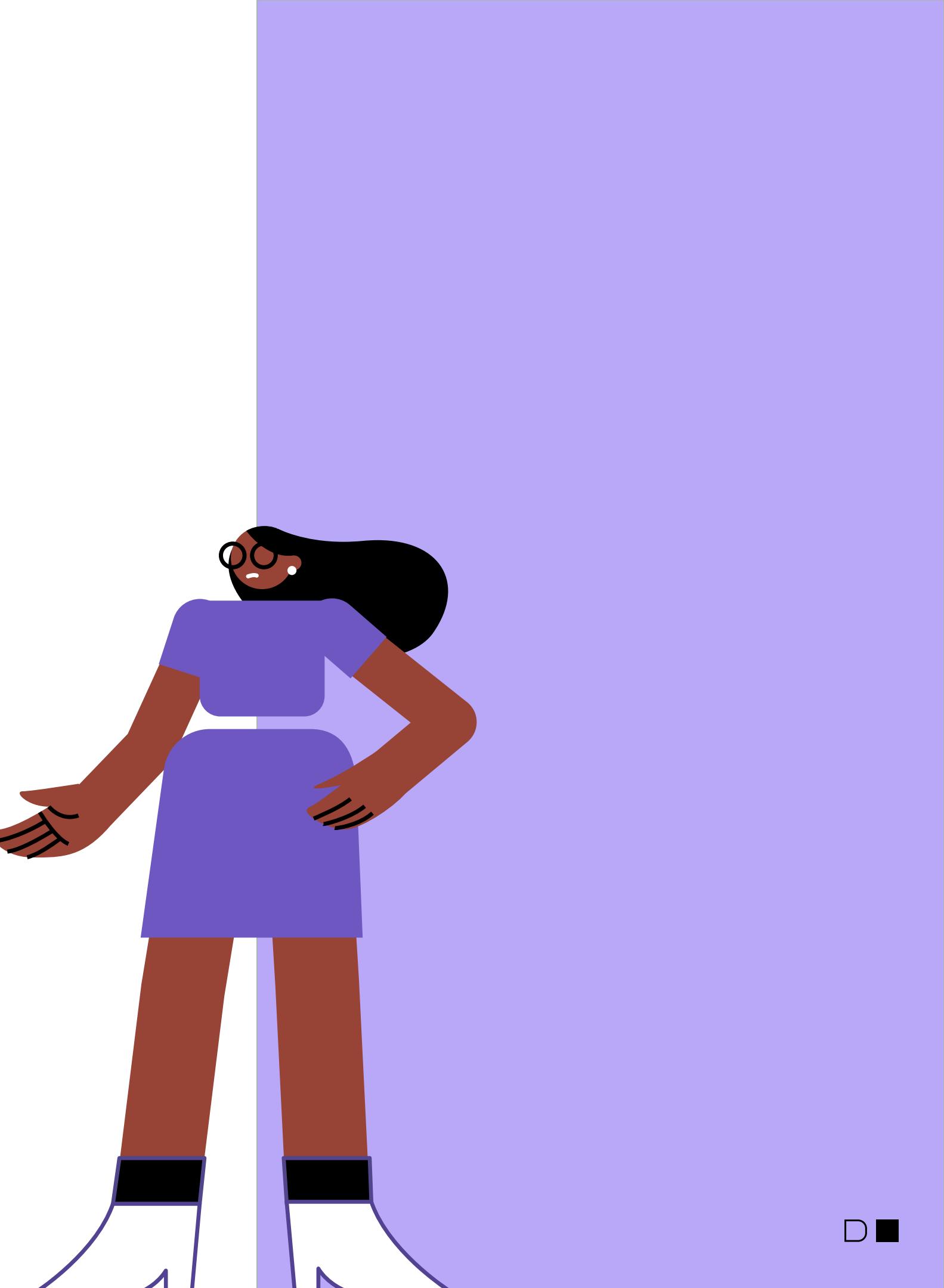


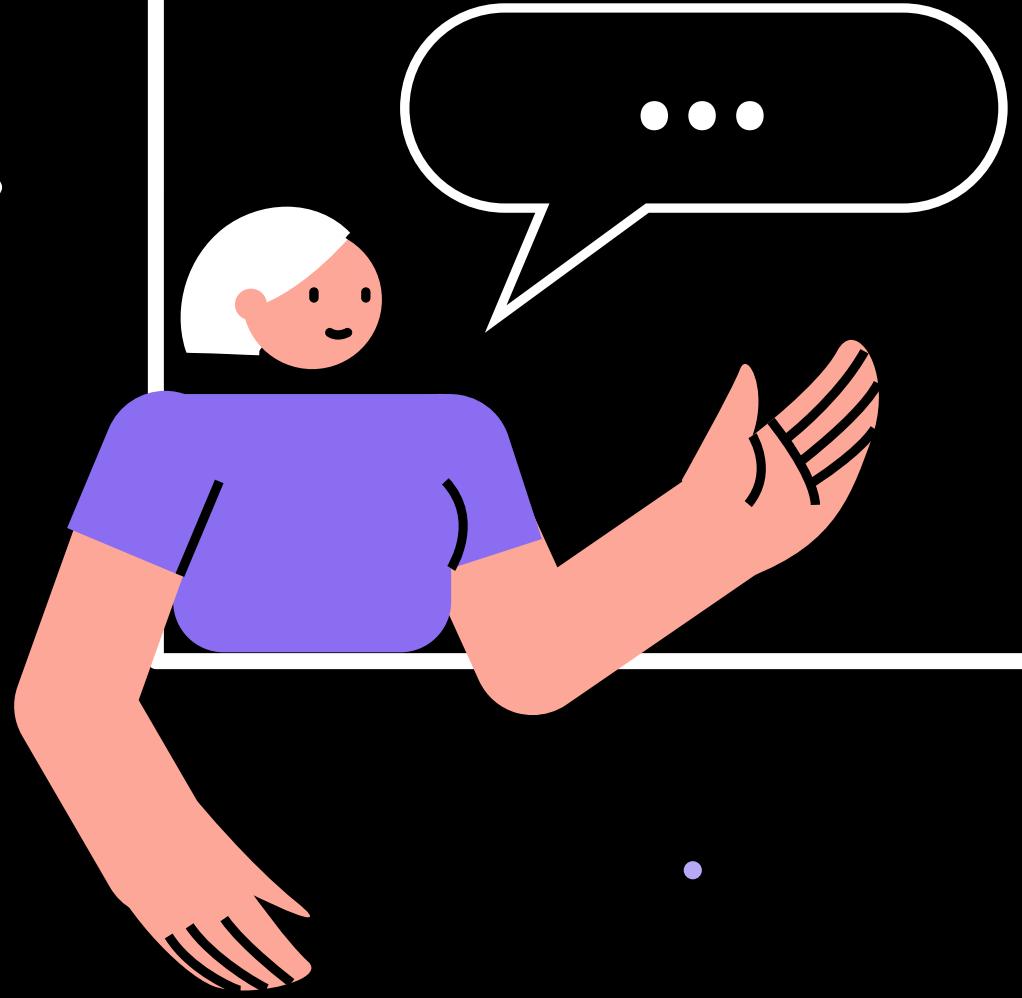
# Conclusiones

En general, cuando escribimos código debemos seguir buenas prácticas para evitar tener un código poco flexible, de difícil mantenimiento y poco extensible. Esto aplica tanto para el código de desarrollo de la aplicación como el código que es parte de pruebas automatizadas.

Los patrones son parte de éstas buenas prácticas que debemos aplicar para evitar esos problemas comunes que se generan cuando manejamos grandes volúmenes de código.

Recordemos que no conviene reinventar la rueda, mejor usemos soluciones ya probadas.





“Un framework define un conjunto de reglas, pautas o mejores prácticas que podemos seguir de manera sistemática para lograr los resultados deseados”.

# Generalidades

Un framework de automatización de pruebas es un **conjunto de pautas** —como estándares de codificación, manejo de datos de prueba, tratamiento de repositorios de objetos, etc.— que, cuando se siguen durante la secuencia de comandos de automatización, traen beneficios tales como una mayor reutilización de código, mayor portabilidad, menor costo de mantenimiento de secuencias de comandos, etc.

Son pautas y no reglas en el sentido estricto de la palabra, ya que no son obligatorios. Se puede escribir un script sin seguir las pautas, pero se perderán las ventajas de tener un framework.

02

## Tipos de frameworks

# Tipos de frameworks para automatización de pruebas

01 Framework de prueba modular

02 Framework de secuencias de comandos línea

03 Framework de prueba basado en datos

04 Framework de prueba basado en palabras clave

05 Framework de prueba híbrido

06 Framework de prueba de desarrollo guiado por el comportamiento

# Framework de prueba modular

En el framework de prueba modular, los probadores crean módulos de scripts de prueba dividiendo la aplicación completa bajo prueba en pruebas independientes más pequeñas.

En palabras simples, los evaluadores dividen la aplicación en varios módulos y crean scripts de prueba individualmente. Estos se pueden combinar para hacer scripts de prueba más grandes utilizando uno maestro para lograr los escenarios requeridos. Este script maestro se utiliza para invocar los módulos individuales para ejecutar escenarios de prueba de un extremo a otro (E2E).

La razón principal para usar este framework es construir una capa de abstracción para proteger el módulo maestro de cualquier cambio realizado en las pruebas individuales.

## Ventajas

# Framework de prueba modular

- Mejor escalabilidad y más fácil de mantener debido a la división de la aplicación completa en diferentes módulos.
- Puede escribir scripts de prueba de forma independiente.
- Los cambios en un módulo tienen un impacto nulo o bajo en los otros módulos.



## Desventajas

# Framework de prueba modular

- Toma más tiempo analizar los casos de prueba e identificar flujos reutilizables.
- Debido a los datos codificados en los scripts de prueba no es posible demandar varios conjuntos de datos.
- Requiere habilidades de codificación para configurar el framework.



# Framework de secuencias de comandos lineal

Conocido como **grabación y reproducción** es el más simple de todos los frameworks de automatización de pruebas ya que el tester registra manualmente cada paso —navegación y entradas del usuario— e inserta puntos de control —pasos de validación— en la primera iteración. Luego, reproduce el script grabado en las iteraciones siguientes.



Ejemplos: Selenium GRID, UFT, Test Complete, Sahi.



## Ventajas

# Framework de secuencias de comandos lineal

- La forma más rápida de generar un script.
- No se requiere experiencia en automatización.
- La forma más sencilla de conocer las funciones de la herramienta de prueba.



## Desventajas

# Framework de secuencias de comandos lineal

- Poca reutilización de guiones.
- Los datos de prueba están codificados en el script.
- Pesadilla de mantenimiento.



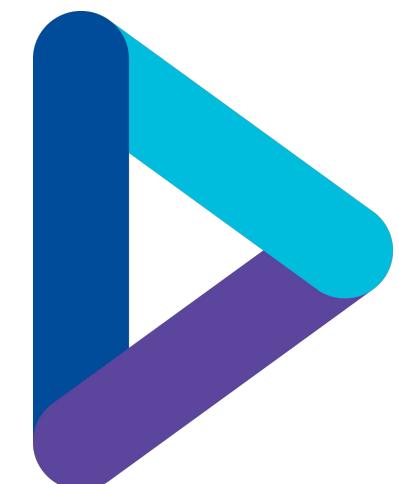
# Framework de prueba basado en datos

En este framework, mientras que la lógica del caso de prueba reside en los scripts de prueba, los datos de prueba se separan y se mantienen fuera de los scripts de prueba.

Los datos de prueba se leen de los archivos externos —archivos de Excel, archivos de texto, archivos CSV, fuentes ODBC, objetos DAO, objetos ADO— y se cargan en las variables dentro del script de prueba.

Las variables se utilizan tanto para los valores de entrada como para los valores de verificación. Los scripts de prueba se preparan utilizando Linear Scripting o Test Library Framework.

Ejemplo: UFT, Specflow (C#)



## Ventajas

# Framework de prueba basado en datos

- Los cambios en los scripts de prueba no afectan los datos de prueba.
- Los casos de prueba se pueden ejecutar con múltiples conjuntos de datos.
- Se puede ejecutar una variedad de escenarios de prueba simplemente variando los datos de prueba en el archivo de datos externo.



## Desventajas

# Framework de prueba basado en datos

- Se necesita más tiempo para planificar y preparar tanto los scripts de prueba como los datos de prueba.



# Framework de prueba basado en palabras clave

Es conocida como prueba basada en tablas o prueba basada en palabras de acción. El desarrollo de este framework requiere tablas de datos y palabras clave, independientemente de la herramienta de automatización de pruebas utilizada para ejecutarlas. Las pruebas se pueden diseñar con o sin la aplicación.

En una prueba basada en palabras clave, la funcionalidad de la aplicación bajo prueba se documenta en una tabla, así como en instrucciones paso a paso para cada prueba.

Aunque trabajar en este framework no requiere muchas habilidades de programación, la configuración inicial —implementarlo— requiere más experiencia.

Ejemplo: Robot Framework (Python).



## Ventajas

# Framework de prueba basado en palabras clave

- Proporciona una alta reutilización de código.
- Herramienta de prueba independiente.
- Aunque la aplicación cambie, los scripts de prueba no cambian.
- Las pruebas se pueden diseñar con o sin la aplicación bajo prueba



## Desventajas

# Framework de prueba basado en palabras clave

- Dado que la inversión inicial es bastante alta, los beneficios de esto solo se pueden obtener si la aplicación es considerablemente grande y los scripts de prueba deben mantenerse durante bastantes años.
- Se requiere una gran experiencia en automatización para crear el framework basado en palabras clave.



# Framework de prueba híbrido

El framework de automatización de pruebas híbridas es la combinación de dos o más frameworks mencionados anteriormente.

Intenta aprovechar las fortalezas y los beneficios de otros frameworks para el entorno de prueba particular que administra.

La mayoría de los equipos están construyendo este framework impulsado por híbridos en el mercado actual. La industria generalmente utiliza el framework de palabras clave en una combinación con el de descomposición de funciones.

Ejemplo: Karate-DSL, Citrus, etc.



# Framework de prueba de desarrollo guiado por el comportamiento

El propósito de este framework de desarrollo guiado por el comportamiento es crear una plataforma que permita a todos —como analistas de negocios, desarrolladores, probadores, etc.— participar activamente. Requiere una mayor colaboración entre los equipos de desarrollo y prueba, pero no requiere que los usuarios estén familiarizados con un lenguaje de programación. Se utiliza un lenguaje natural no técnico para crear especificaciones de prueba —Gherkin—.

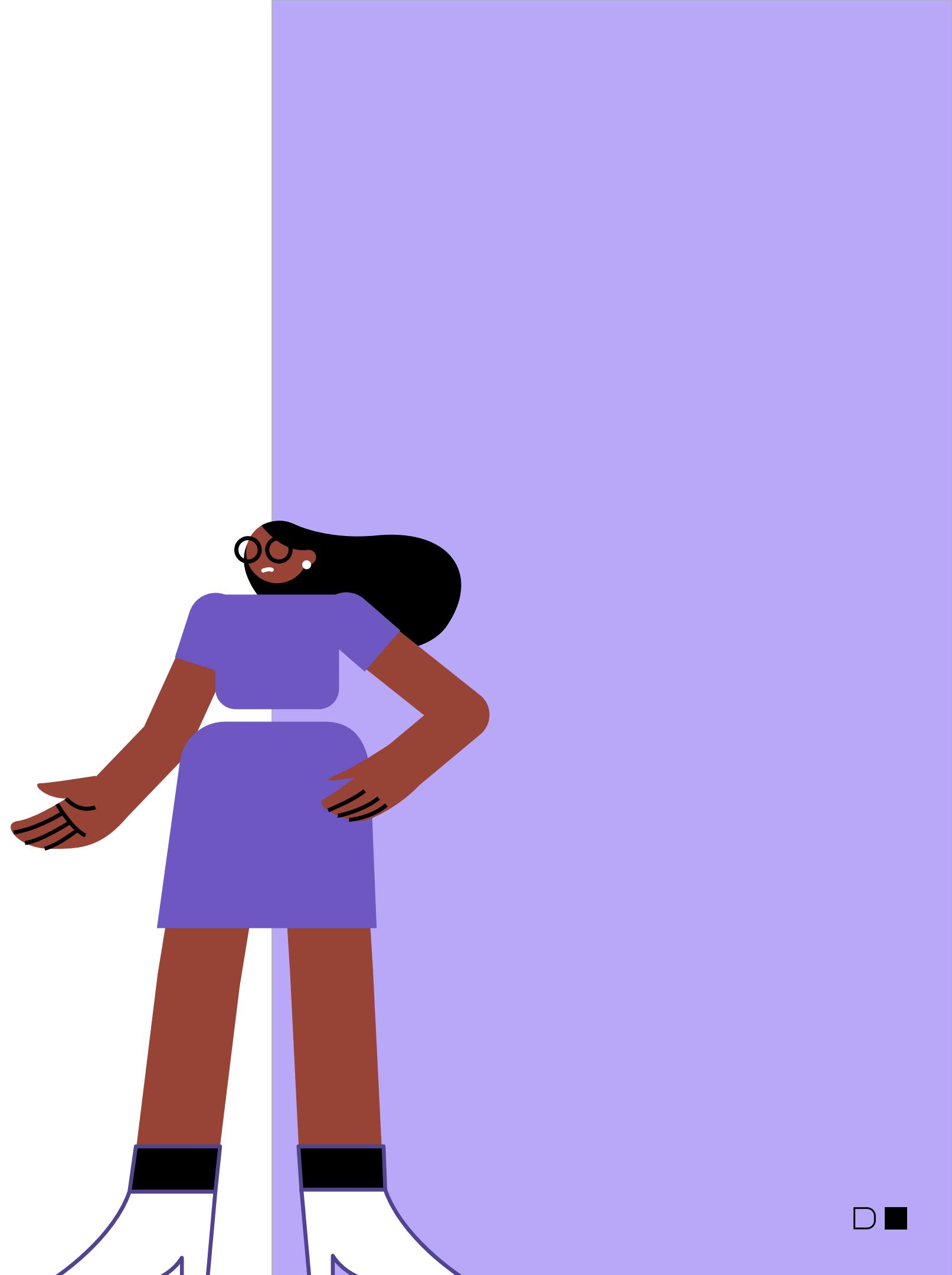
Algunas de las herramientas disponibles en el mercado para el desarrollo impulsado por el comportamiento son JBehave (Java) , Cucumber , Specflow (C#), Serenity, etc.



# Conclusiones

Estos son los beneficios generales de contar con un framework de prueba:

- Ayuda a reducir el riesgo y los costos.
- Mejora la eficiencia de las pruebas.
- Ayuda a reducir el costo de mantenimiento.
- Permite la reutilización de código.
- Permite alcanzar la máxima cobertura de prueba.
- Maximiza la funcionalidad de la aplicación.
- Ayuda a reducir la duplicación de casos de prueba.
- Ayuda a mejorar la eficiencia y el rendimiento de la prueba con la automatización de la prueba.



# Partes de un framework para automatización de pruebas

Como testers, todos sabemos que para probar una aplicación basada en la web, necesitamos realizar acciones específicas (como hacer clic, escribir, etc.) en los elementos HTML. Ahora, cuando buscamos la automatización de esas aplicaciones, la herramienta también debería ser capaz de realizar las mismas operaciones en los elementos HTML que un ser humano podría hacer. Entonces, surge la pregunta, ¿cómo realizamos estas acciones? La respuesta a esto es mediante los siguientes elementos:



## Locators

¿Cómo identifican estas herramientas de automatización en qué elemento HTML necesitan realizar una operación en particular? La respuesta a esto es "Locators". Los localizadores son la forma de identificar un elemento HTML en una página web, y casi todas las herramientas de automatización brindan la capacidad de usar localizadores para la identificación de elementos HTML en una página web. Existen varios métodos para realizar la identificación de los elementos HTML, como por ejemplo por ID, Name, CSS Selector, ClassName, TagName, Linktext, etc. Trabajaremos con todos estos métodos durante la cursada.

## Helpers

¿Existe dentro de las herramientas de automatización alguna funcionalidad que simule el comportamiento humano de las pruebas manuales ? La respuesta a esto es "Helpers".

Los helpers son pequeñas piezas de funcionalidad que realizan interacciones de usuario simples que se pueden reutilizar. Permiten usar funcionalidad en sus pruebas para las que normalmente necesitaría habilidades de codificación. A estos helpers se le pueden enviar diferentes variables o parámetros y realizarán una funcionalidad deseada.

## Before Test & After Test

@BeforeTest y @AfterTest son métodos de los framework de automatización que nos permiten ejecutar funcionalidades antes y después de los test. Para marcos como pruebas de humo, por ejemplo, @BeforeTest se usa para crear una configuración de datos iniciales antes de ejecutar el set de test, mientras que @AfterTest se puede utilizar para enviar los resultados de los test por correo electrónico luego de su ejecución.

## Before Execution & After Execution

@BeforeExecution y @AfterExecution son métodos de los framework de automatización que nos permiten ejecutar funcionalidades antes y después de la ejecución. Por ejemplo, pueden ser utilizados para abrir y cerrar un navegador web./span>

## Common Functions

Los framework de automatización cuentan con diferentes funciones ya incorporadas que nos permiten realizar diferentes funcionalidades sin necesidad de escribir código. Como por ejemplo: click(), size(), get(), etc.

## TestSet (scripts) & TestData (Data)

Los frameworks de automatización nos permiten separar los casos de pruebas (scripts) y los datos a probar (data) en diferentes set. Esto nos permite tener un TestSet en donde vamos a encontrar todas las pruebas a ejecutar y un TestData en donde podemos definir los datos a probar en diferentes formatos.

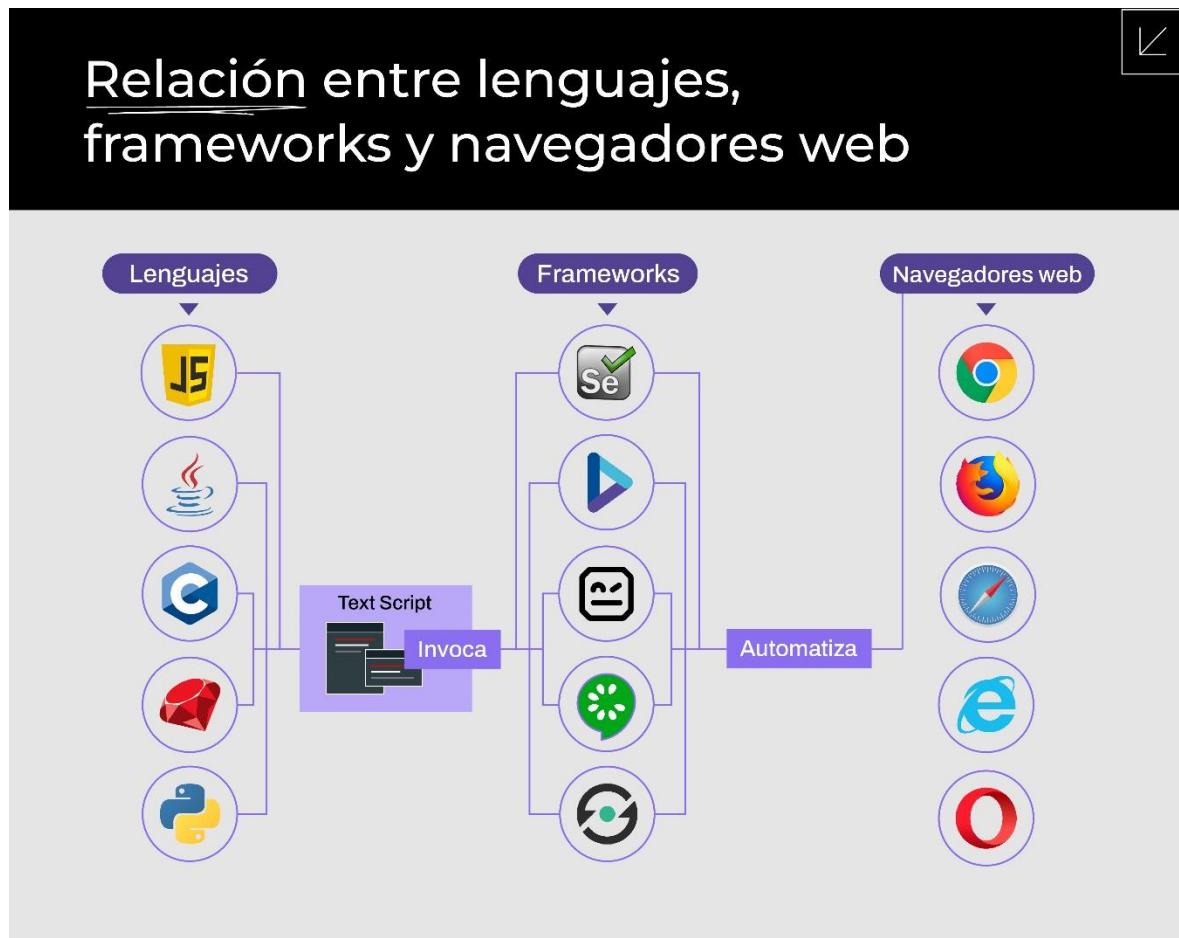
Dado que los casos de prueba están separados del conjunto de datos, se puede modificar fácilmente el caso de prueba de una funcionalidad en particular sin realizar cambios en el código.

## Reports

Los frameworks de automatización nos permiten generar diferentes tipos de reportes, los cuales nos servirán a la hora de presentar métricas, analizar los resultados, etc.

## CI/CD

Los frameworks de automatización nos permiten integrar CI/CD con herramientas como Jenkins. De esta manera, por ejemplo, podemos ejecutar nuestros test cada vez que se realice un merge en la rama maestra y verificar que nada se rompió.



# Comandos Selenium: Elementos web, Navegador, navegación, espera, cambio.

## Elementos Web

Sintaxis	Uso	Ejemplo
.findElement()	Encuentra un elemento dado mediante el uso de localizadores (locators).	driver.findElement(By.id("Button"));
.clear()	Borra los valores dentro de un elemento de entrada de texto.	driver.findElement(By.xpath("//input[@name='name']")).clear();
.click()	Da clic a un elemento dado.	driver.findElement(By.id("Button")).click();
.getAttribute()	Obtiene el valor de un atributo.	driver.findElement(By.id("Button")).getAttribute("name");
.getLocation()	Obtiene la ubicación del elemento en la página.	driver.findElement(By.name("image")).getLocation();
.getSize()	Obtiene el ancho y el alto del elemento renderizado.	driver.findElement(By.name("icon")).getSize();
.getTagName()	Obtiene el nombre de la etiqueta de un elemento.	driver.findElement(By.id("icon")).getTagName();
.getText()	Obtiene el texto del elemento dado y devolverá un valor de tipo String.	driver.findElement(By.id("msj-error")).getText();
.isDisplayed()	Obtiene un valor booleano al determinar si un elemento se muestra o no.	driver.findElement(By.id("image")).isDisplayed();
.isEnabled()	Obtiene un valor booleano al determinar si un elemento está habilitado o no.	driver.findElement(By.id("image")).isEnabled();
.sendKeys()	Ingrresa valores a una entrada de texto.	driver.findElement(By.id("user")).sendKeys("profes@digitalhouse.com");
.submit()	Envía información de mejor manera que un clic en algún formulario.	driver.findElement(By.id("login")).submit;

# Navegador

Sintaxis	Uso	Ejemplo
.get()	Abre la página web indicada.	driver.get("https://digitalhouse.com");
.getCurrentUrl()	Obtiene la URL de la página actual como un valor de tipo String.	driver.getCurrentUrl();
.getTitle()	Obtiene el título de la página actual como un valor de tipo String.	driver.getTitle();
.close()	Cierra la ventana actual abierta por el webdriver.	driver.close();
.quit()	Cierra todas las ventanas abiertas por el webdriver.	driver.quit();
.manage().window().maximize()	Maximiza la ventana actual del navegador.	driver.manage().window().maximize();
.manage().window().fullscreen()	Muestra en pantalla completa la ventana actual —si aún no está en pantalla completa—.	driver.window().fullscreen();
.manage().window().getPosition()	Obtiene la posición de la ventana actual.	driver.manage().window().getPosition();
.manage().window().setPosition()	Define la posición de la ventana actual.	driver.manage().window().setPosition(750,900);
.manage().window().setSize()	Define el tamaño de la venta actual.	driver.manage().window().setSize(750,900);
.manage().deleteCookie(-Cookie cookie)	Borra una cookie específica.	driver.manage().deleteCookie(arg0);
manage().deleteAllCookies()	Borra todas las cookies.	driver.manage().deleteAllCookies();
.manage().getCookies()	Obtiene todas las cookies disponibles.	driver.manage().getCookies();
.getWindowHandle()	Obtiene el nombre de una ventana dada.	driver.getWindowHandles();

## Navegación

Sintaxis	Uso	Ejemplo
.back()	Indica al navegador que se redirija a la página web inmediatamente anterior.	driver.navigate(). <b>back()</b> ;
.forward()	Indica al navegador que se redirija a la página web inmediatamente posterior.	driver.navigate(). <b>forward()</b> ;
.refresh()	Indica al navegador que se actualice/recargue la pagina web donde está situado.	driver.navigate(). <b>refresh()</b> ;
.to()	Indica al navegador que se dirija a la pagina web indicada.	driver.navigate(). <b>to</b> ("https://digitalhouse.com");

## Espera

Sintaxis	Uso	Ejemplo
.pageLoadTimeout(Número de tiempo, TimeUnit. Unidad de tiempo)	Establece la cantidad de tiempo de espera para que se complete la carga de una página antes de arrojar un error.	driver.manage().timeouts(). <b>pageLoadTimeout</b> (5, <b>TimeUnit.SECONDS</b> );
.implicitlyWait(Número de tiempo, TimeUnit.Unidad de tiempo)	Establece la cantidad de tiempo de espera de un elemento.	driver.manage().timeouts(). <b>implicitlyWait</b> (10, <b>TimeUnit.SECONDS</b> )
WebDriverWait	Pausa la ejecución hasta agotar el tiempo o se cumpla una condición esperada usando la clase WebDriverWait.	<b>WebDriverWait</b> wait = new WebDriverWait(- driver,8);
FluentWait	Se considera el núcleo de la espera explícita porque WebDriverWait extiende FluentWait.	Wait wait = new <b>FluentWait</b> <WebDriver>(driver) .withTimeout(45, TimeUnit.SECONDS) .pollingEvery(5, TimeUnit.SECONDS) .ignoring(NoSuchElementException.class);

## Cambio

Sintaxis	Uso	Ejemplo
.switchTo().window()	Cambia de ventana siempre que se indique el nombre de esta.	driver. <b>switchTo().window</b> ("Playground");
.switchTo().alert()	Cambia a un pop up, pudiendo seleccionar una de las opciones disponibles tales como aceptar, cerrar, cancelar, etc.	driver. <b>switchTo().alert().accept();</b>
.switchTo().frame()	Cambia un frame o iframe siempre que se indique el nombre de este.	driver. <b>switchTo().frame</b> ("frameName");

# Los atributos de la clase Artículo

Necesitamos crear una clase **Artículo**, la información con la que contamos es:

- Un Artículo tiene una descripción
- Tiene un precio de venta
- Tiene un stock

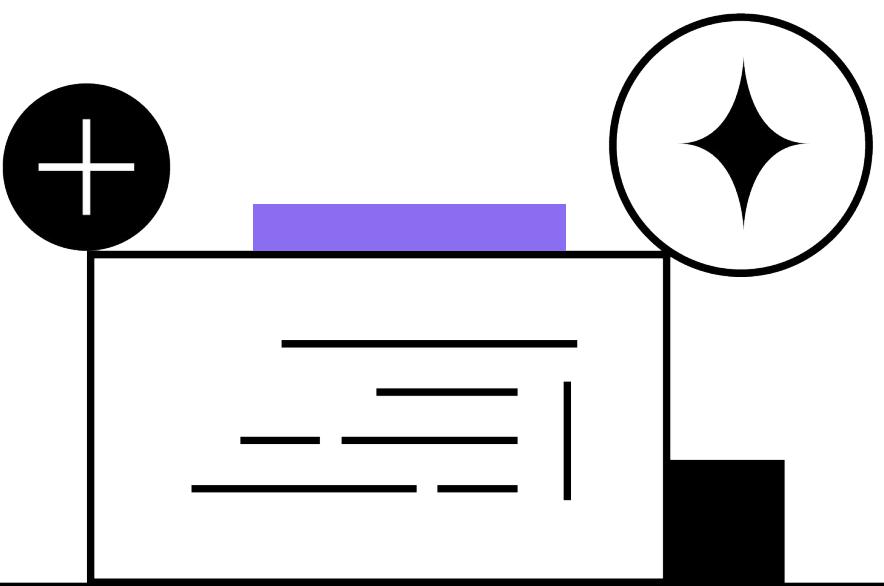
Esta clase debe poder responder si hay stock y cuál es su precio.

Diseño de la clase Artículo

Artículo

- descripcion :String
- precioVenta: double
- stock: int

- + Articulo(descripcion:String, cantidad:int, precio:double)
- + boolean hayStock()
- + double consultarPrecio()



02

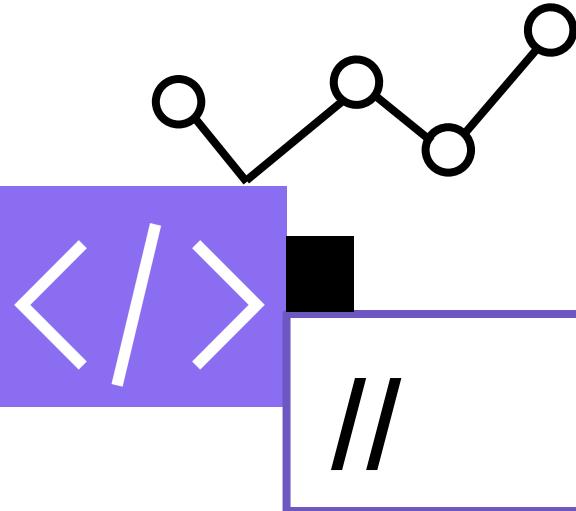
## Modificadores de acceso

# Accediendo a nuestra clase Artículo

Para definir cuándo un atributo, método o constructor puede ser accesible o no, usamos las palabras reservadas: **Public, Private o Protected**

Entonces, al usar este concepto aplicado a nuestra clase artículo **es necesario que sus atributos tengan un modificador de acceso de tipo Private** ya que nos interesa que el código solo sea accesible dentro de la clase declarada.

Si necesitamos acceder a ellos para ver o cambiar tales atributos necesitamos agregar métodos de acceso setters y getters que veremos con detalle mas adelante.



# Los atributos de la clase Articulo en Java

```
public class Articulo{  
    private String descripcion;  
    private double precioVenta;  
    private int stock;  
}
```

03

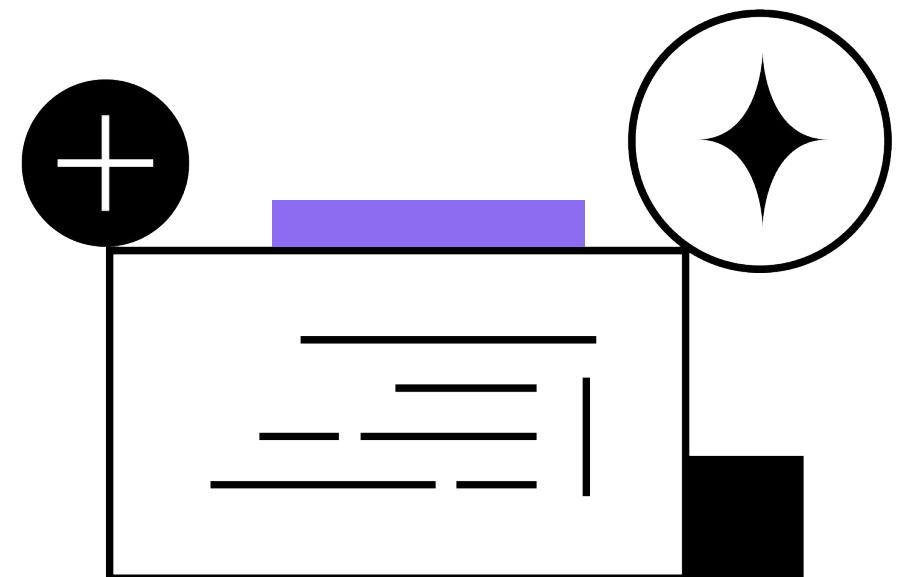
# Constructores

# Constructores en la clase Artículo

## Artículo

- descripción :String
- precioVenta: double
- stock: int

- + Artículo(descripción:String, cantidad:int, precio:double)
- + boolean hayStock()
- + double consultarPrecio()

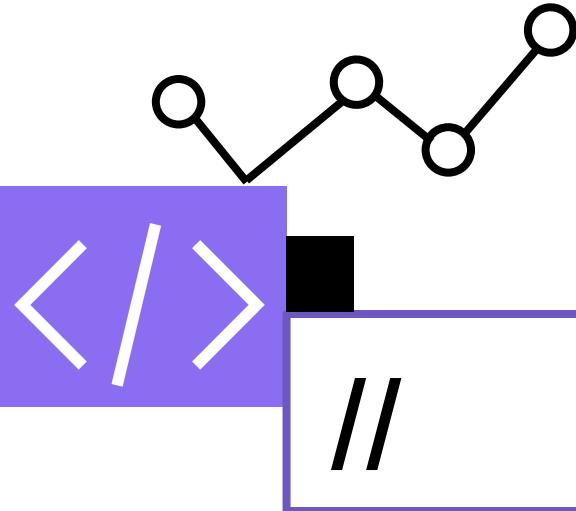


# Los métodos de la clase Articulo en Java



```
...
public class Articulo{
    private String descripcion;
    private double precioVenta;
    private int stock;

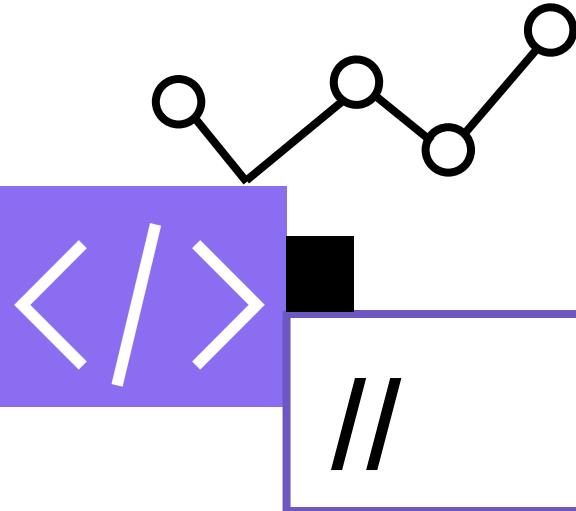
    public Articulo(String descripcion, int cantidad,double
precio){
        this.descripcion=descripcion;
        precioVenta=precio;
        stock=cantidad;
    }
}
```



# {código}

```
public class Articulo{  
    private String descripcion;  
    private double precioVenta;  
    private int stock;  
  
    public Articulo(String descripcion, int  
cantidad, double precio){  
  
        this.descripcion=descripcion;  
        precioVenta= precio;  
        stock=cantidad;  
    }  
}
```

El constructor es un método que no tiene tipo de dato, se llama igual que la clase. Recibe como parámetros los valores que se desea asignar inicialmente a los atributos, es decir, los valores iniciales. Se puede usar para inicializar los atributos.



# {código}

```
public class Articulo{  
    private String descripcion;  
    private double precioVenta;  
    private int stock;  
  
    public Articulo(String descripcion, int  
cantidad, double precio){  
        this.descripcion=descripcion;  
        precioVenta= precio;  
        stock=cantidad;  
    }  
}
```

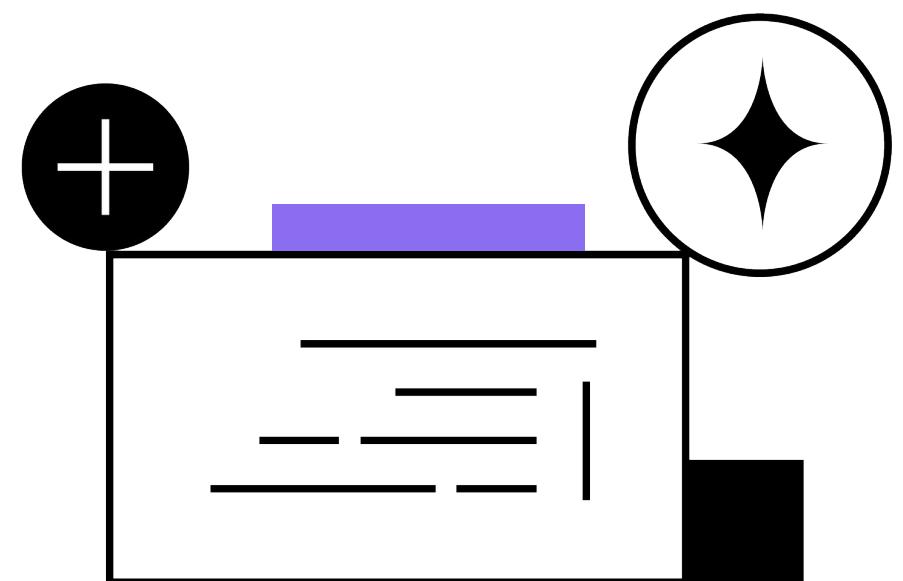
Para diferenciar el atributo `descripcion` del parámetro que tiene el mismo nombre, usamos **this**. **This** hace referencia al objeto o instancia con la que se está trabajando.

# Los métodos de la clase Articulo

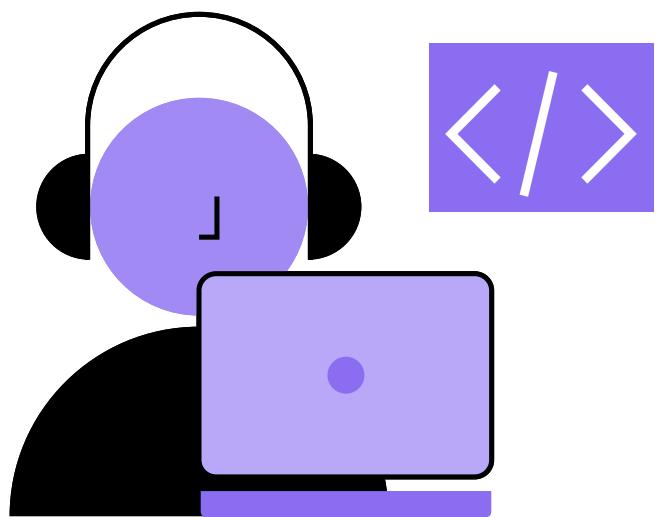
## Artículo

- descripcion :String
- precioVenta: double
- stock: int
  
- + Articulo(descripcion:String,  
cantidad:int, precio:double)
- + boolean hayStock()
- + double consultarPrecio()

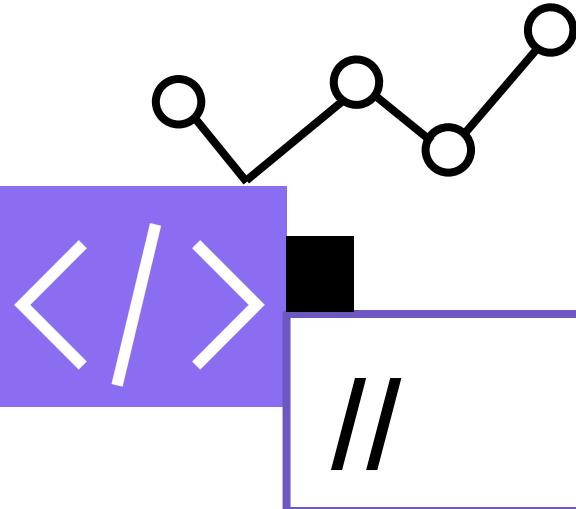
Los métodos de  
nuestra clase son  
hayStock() y  
consultarPrecio()



# Los métodos de la clase Articulo en Java



```
...  
  
public class Articulo{  
    private String descripcion;  
    private double precioVenta;  
    private int stock;  
  
    + public Articulo(String descripcion, int cantidad,double precio)  
  
    public boolean hayStock(){  
        return stock>0;  
    }  
    public double consultarPrecio(){  
        return precioVenta;  
    }  
}
```



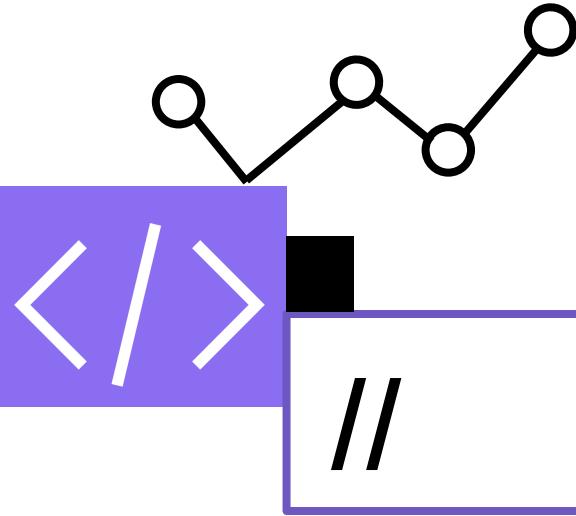
# {código}

```
public class Articulo
    private String descripcion;
    private double precioVenta;
    private int stock;
+ public Articulo(String descripcion, int
cantidad, double precio)

public boolean hayStock(){
    return stock>0; }

public double consultarPrecio()
    return precioVenta;
```

Método hayStock()  
Devuelve **true** si stock es  
mayor a 0 y **false** en caso  
contrario



# {código}

```
public class Articulo
    private String descripcion;
    private double precioVenta;
    private double precioCompra;
    private int stock;
+ public Articulo(String descripcion, int
cantidad, double precio)
    public boolean hayStock()
        return stock>0;

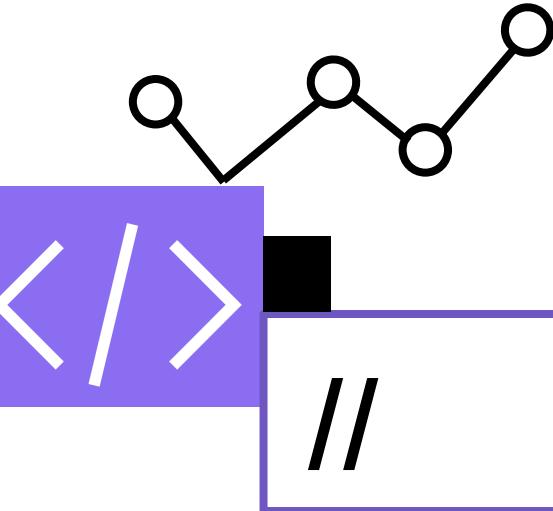
    public double consultarPrecio(){
        return precioVenta;
    }
}
```

Método consultarPrecio()  
Devuelve el **Precio de  
Venta**

# Setters y Getters

Para lograr y proteger el encapsulamiento de nuestros atributos en la clase articulo debemos definirlos como **privados** y agregar getters y setters.

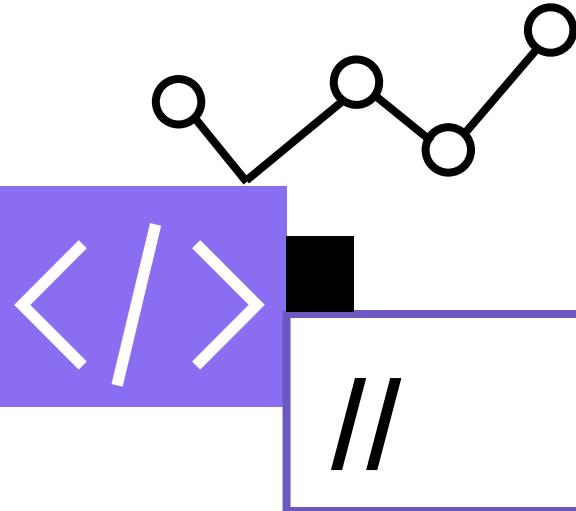
Recuerden que es una buena práctica impedir que cualquier otro objeto pueda tener acceso a la estructura interna de un objeto, es decir, preservar el **encapsulamiento**.



# {código}

```
public String getDescripcion(){
    return descripcion;}
public double getPrecioVenta(){
    return precioVenta;}
public int getStock(){
    return stock;}
public void setDescripcion(String
descripción){
    this.descripcion= descripción;}
public void setPrecioVenta(double precio){
    precioVenta=precio;}
public void setStock(int stock){
    this.stock=stock;}
```

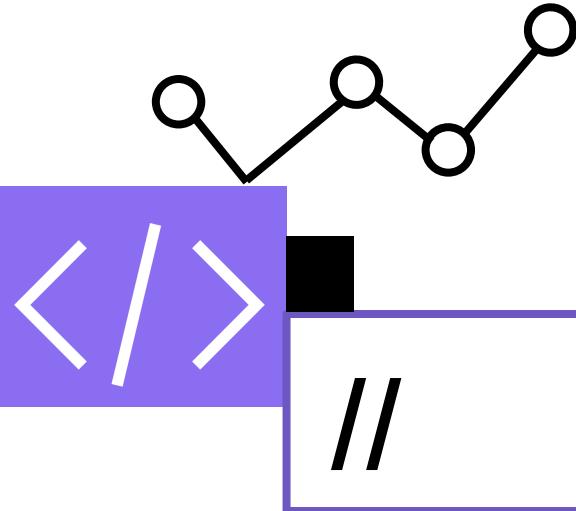
Los métodos **get** permiten acceder al valor de un atributo para una consulta o para usar ese valor en otra operación.



# {código}

```
public String getDescripcion(){
    return descripcion;
}
public double getPrecioVenta(){
    return precioVenta;
}
public int getStock(){
    return stock;
}
public void setDescripcion(String descripcion){
    this.descripcion= descripcion;
}
public void setPrcioVenta(double precio){
    precioVenta=precio;
}
public void setStock(int stock){
    this.stock=stock;
}
```

Los métodos **set** permiten cambiar el valor de un atributo, reciben por parámetro el nuevo valor y lo asignan al atributo correspondiente.



# {código}

```
public class Articulo{  
    private String descripcion;  
    private double precioVenta;  
    private int stock;  
  
    public Articulo(String descripcion, int  
cantidad, double precio){  
        this.descripcion=descripcion;  
        precioVenta= precio;  
        stock=cantidad;  
    }  
}
```

Para diferenciar el atributo **descripción** del parámetro que tiene el mismo nombre, usamos **this**. This hace referencia al objeto o instancia con la que se está trabajando.

# No olviden que:

- Cuando definamos un objeto, debemos dejar sus atributos privados.
- Los métodos que sean públicos serán vistos por los otros objetos.
- Usar siempre métodos públicos para ver o modificar las características de los objetos.
- Para cambiar el valor de un atributo se usa un método set, por ejemplo, para cambiar el nombre será `setNombre(String)`
- Para obtener el valor de un atributo se usa un método get, por ejemplo, para saber el nombre será `getNombre(): String`

Los atributos, métodos y constructores de nuestra clase Articulo se verían así:

```
public class Articulo{  
    private String descripcion;  
    private double precioVenta;  
    private int stock;  
    public Articulo(String descripcion, int cantidad,double precio)  
    {  
        this.descripcion=descripcion;  
        precioVenta=precio;  
        stock=cantidad;  
    }  
  
    public String getDescripcion(){  
        return descripcion;}  
    public double getPrecioVenta(){  
        return precioVenta;}  
    public int getStock(){  
        return stock;}  
    public void setDescripcion(String descripcion)  
    {  
        this.descripcion= descripcion;}  
    public void setPrcioVenta(double precio){  
        precioVenta=precio; }
```



# Localizadores en Selenium

Los localizadores se definen como una **dirección que identifica de forma única un elemento web dentro de la página web**. Es un comando que le indica al IDE de Selenium qué elementos de la interfaz gráfica de usuario, como cuadros de texto, botones, casillas de verificación, etc., debe operar. Encontrar los elementos correctos de la GUI es un requisito previo para crear un script de automatización, pero identificar con precisión los elementos de la GUI es mucho más difícil de lo que parece. A veces, incluso se puede acabar trabajando con elementos de la interfaz gráfica de usuario incorrectos o sin ningún elemento. Así, el uso del buscador adecuado garantiza que las pruebas sean más rápidas, más fiables o de menor mantenimiento en comparación con las versiones.

Si tienen la suerte de trabajar con IDs y clases únicas, normalmente lo tienen todo listo. Pero habrá ocasiones en las que elegir el localizador adecuado se convierta en una pesadilla debido a la complejidad de encontrar los elementos de la página web.

## Tipos de localizadores en Selenium

Hay una gama diversa de elementos web como caja de texto, id, botón de opción, etc. Se requiere un enfoque eficaz y preciso para identificar estos elementos. Así, se puede afirmar que con el aumento de la eficacia del localizador, aumentará la estabilidad del script de automatización.

Hay ocho estrategias diferentes de localización de elementos incorporadas en Selenium WebDriver:

<u>Localizador</u>	<u>Descripción</u>	<u>Sintaxis</u>
class name	Buscar elementos cuyo nombre de clase contenga el valor de búsqueda —no se permiten nombres de clase compuestos—.	driver.findElement(By.className (<element class>))
css selector	Buscar elementos que coincidan con un selector CSS.	driver.findElement(By.cssSelector (<css selector>))
id	Encuentra los elementos cuyo atributo ID coincide con el valor de búsqueda.	driver.findElement(By.id (<element ID>))
name	Encuentra los elementos cuyo atributo NAME se	driver.findElement(By.name (<element name>))

	corresponde con el valor de búsqueda.	
link text	Busca elementos de anclaje cuyo texto visible coincide con el valor de búsqueda.	driver.findElement(By.linkText (<linktext>))
partial link text	Encuentra los elementos de anclaje cuyo texto visible contiene el valor de búsqueda. Si hay varios elementos coincidentes, sólo se seleccionará el primero.	driver.findElement(By.partialLinkText (<linktext>))
tag name	Busca elementos cuyo nombre de etiqueta coincide con el valor de búsqueda.	driver.findElement(By.tagName (<htmlltagname>))
xpath	Busca elementos que coincidan con una expresión XPath.	driver.findElement(By.xpath (<xpath>))

## Prácticas recomendadas para los localizadores de Selenium

Entender el concepto de localizadores en Selenium es una cosa, pero saber utilizarlos es otra muy distinta. Para poder **construir un localizador robusto** hay que entender qué es un localizador robusto.

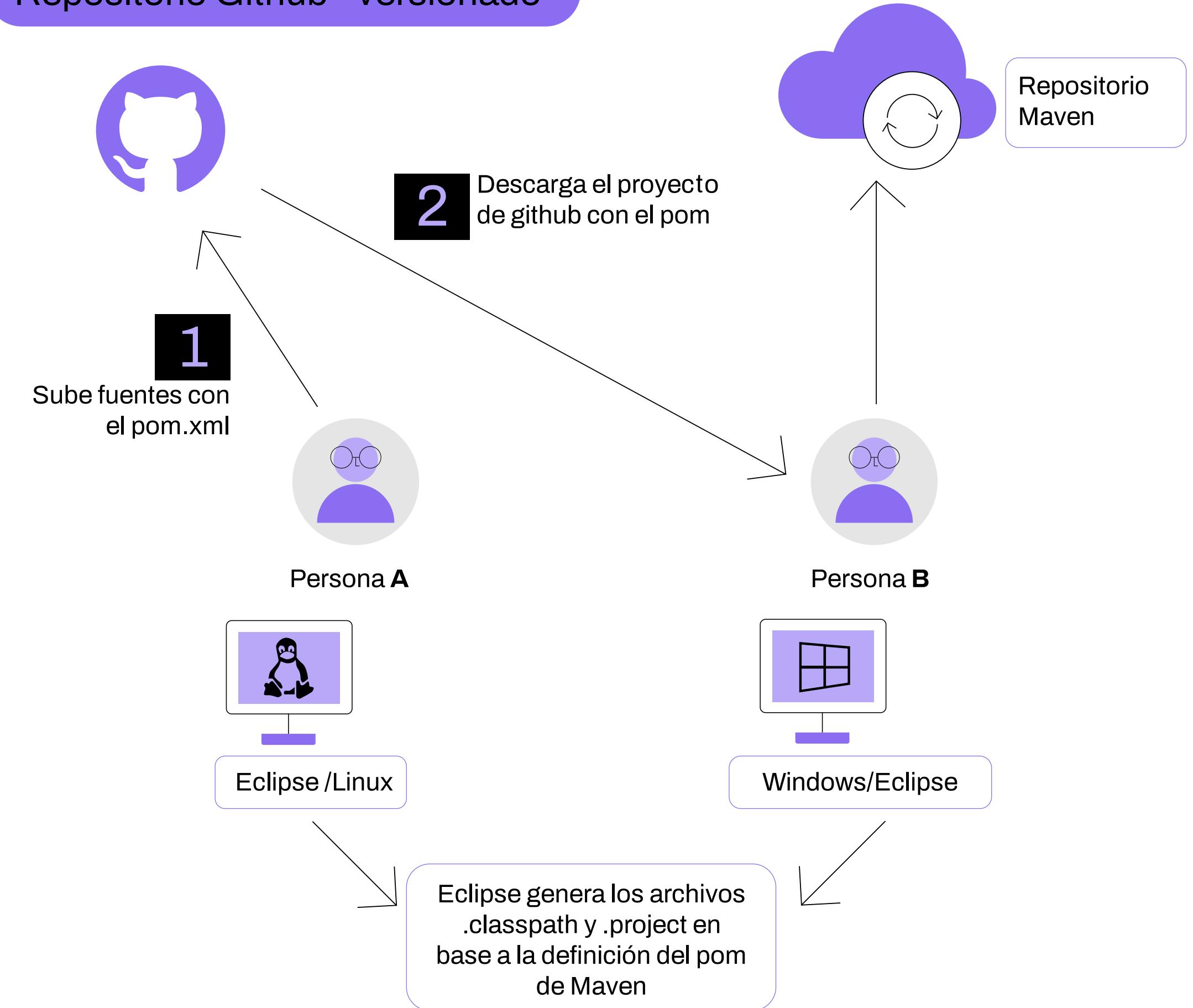
A continuación se enumeran tres criterios que debe seguir cuando utilice localizadores en Selenium:

- Los localizadores robustos en Selenium son tan simples y pequeños como es posible: cuantos más elementos contenga un localizador, mayores serán las posibilidades de que se rompa debido a un cambio en la estructura de la página.
- Los localizadores de Selenium siguen funcionando después de cambiar las propiedades de un elemento de la interfaz de usuario: confiar en los atributos que se cambian con frecuencia, como las clases modificadoras (menu\_item-red), nunca es una buena práctica.
- Los localizadores de Selenium, que son robustos por naturaleza, siguen funcionando después de que se cambien los elementos de la UI alrededor del elemento al que se dirige: siempre que se utilice un atributo no único, lo más probable es que los localizadores se rompan porque alguien haya añadido un elemento con el mismo atributo por encima.

# ¿Cómo funciona Maven?

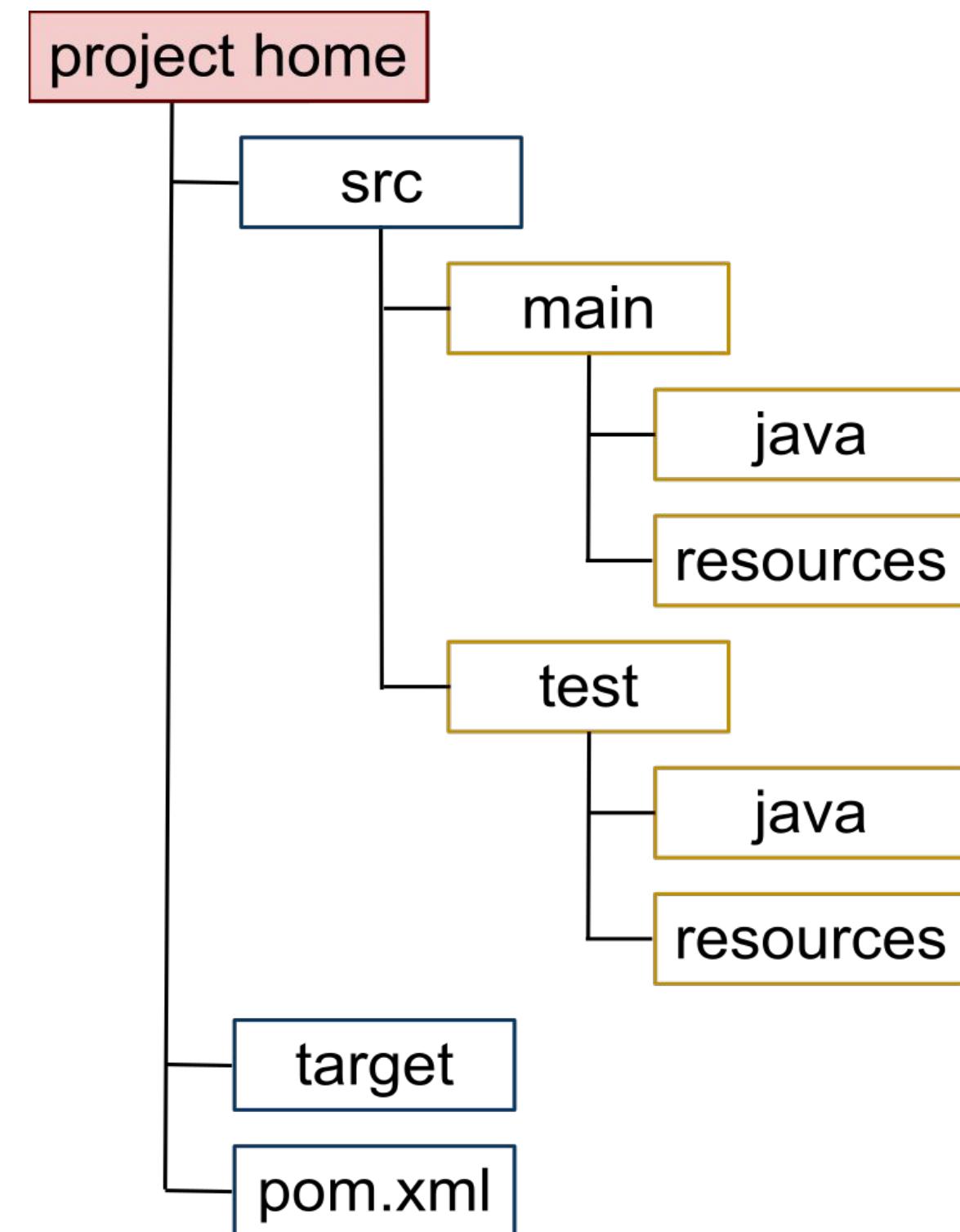
Adaptado de:  
<https://wiki.uqbar.org/wiki/articles/maven.html>

## Repositorio Github - versionado



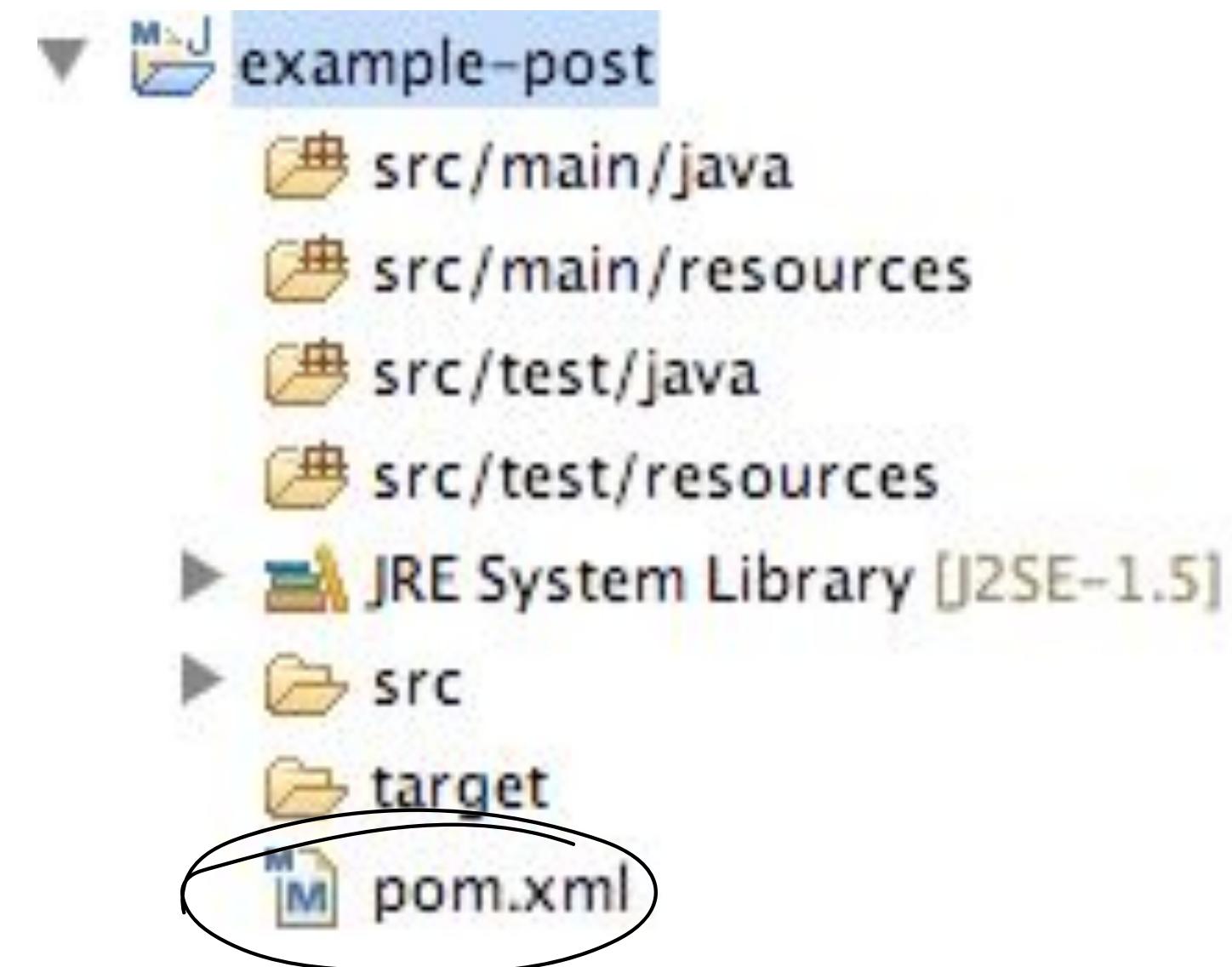
# Estructura de un proyecto Maven

Al crear un proyecto de **Maven**, automáticamente se nos generará una estructura de carpetas muy concreta que ya viene predefinida. Cada uno de los directorios tiene una funcionalidad dentro del proyecto.



# pom.xml

Podemos decir que este archivo es **el corazón del proyecto**. Project Object Model (POM) es el encargado de gestionar y construir los proyectos, contiene el listado de dependencias que son necesarias para que el proyecto funcione, plugins, etc. Toda la información del proyecto está basada en este archivo. Tiene extensión .xml.



# pom.xml

El archivo **pom** con la estructura más básica posible, contendrá:

```
<?xml version="1.0" encoding="UTF-8"?>  
<project>  
  <modelVersion>4.0.0</modelVersion>  
  <groupId>com.mycompany.app</groupId>  
  <artifactId>my-app</artifactId>  
  <version>1</version>  
</project>
```

**Project:** root. Es el esqueleto principal del archivo pom

**ModelVersion:** se debe establecer en 4.0.0. Es la última versión de Maven

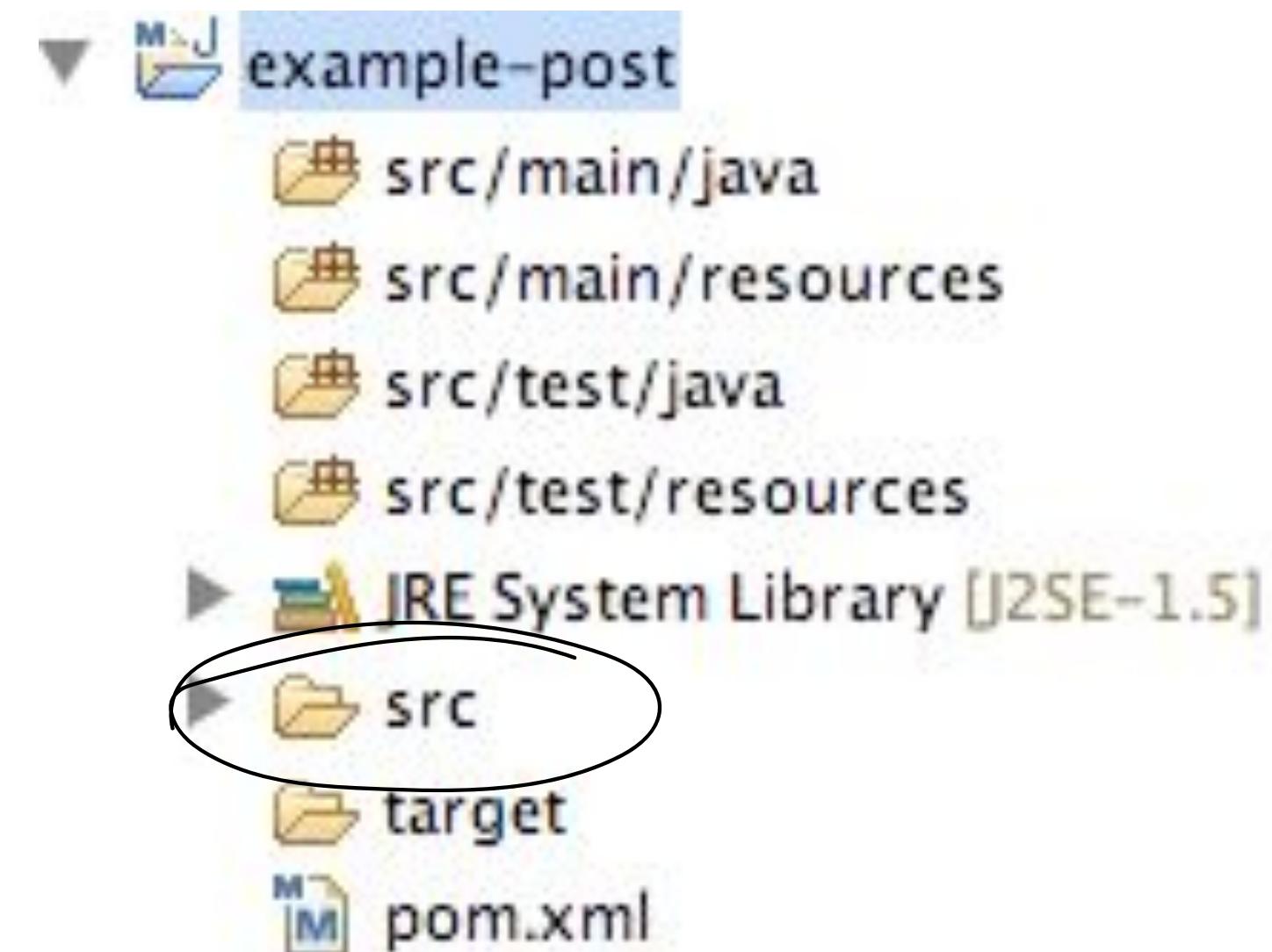
**GroupId:** es el id del grupo del proyecto.

**ArtifactId:** es el id del artefacto, es decir del proyecto.

**Versión:** es la versión del proyecto.

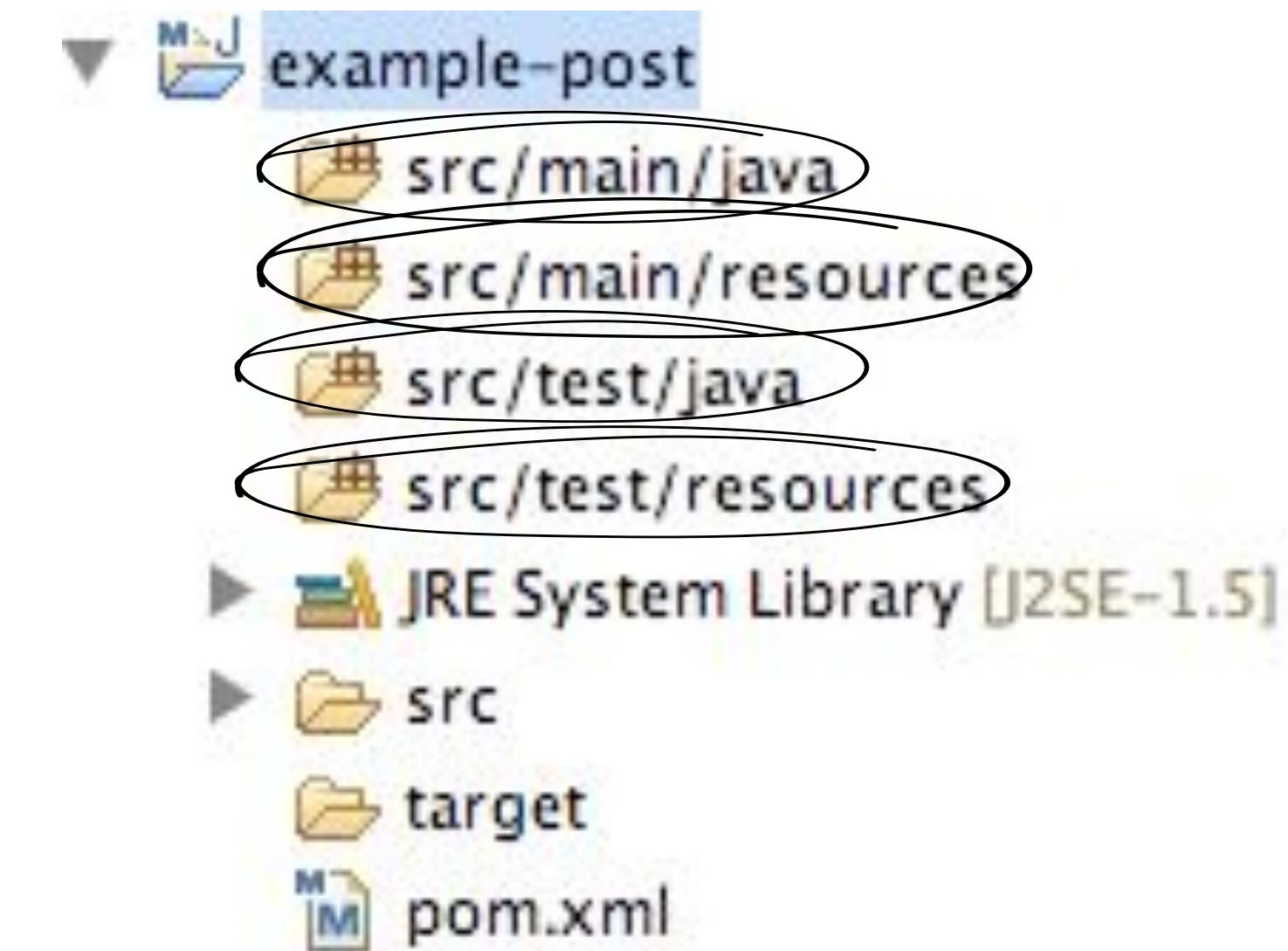
# Directorio src

El directorio **src** contiene todo el material de origen para construir el proyecto, su sitio, sus pruebas, etc.  
Tiene dos subdirectorios principales: **main** y **test**.



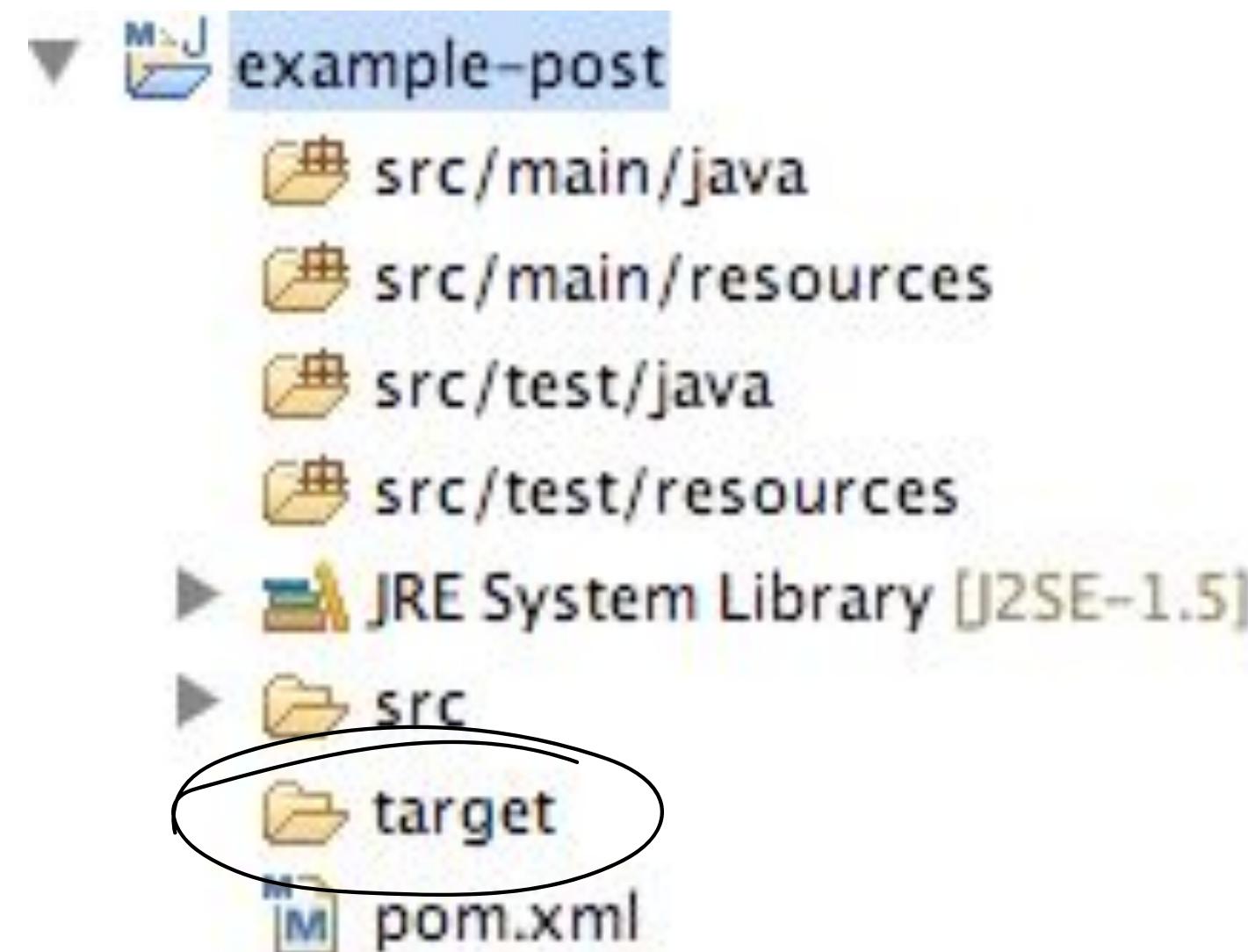
# Directorio src

- **src/main/java:** contiene el código fuente. Aquí escribiremos —o estarán, si el proyecto es cargado— las clases con extensión .java. El contenido de este directorio se conoce bajo el nombre de módulo.
- **src/main/resources:** contiene los recursos estáticos —XML, propiedades, imágenes...— que necesita nuestro módulo para funcionar correctamente.
- **src/test/java:** contiene los ficheros de pruebas —testing— para verificar el correcto funcionamiento del módulo.
- **src/test/resources:** almacena los archivos que genera Maven al usar los comandos —compile, package...—.



# Directorio target

El directorio **target** se utiliza para albergar todos los resultados de la compilación. Incluso contiene los resultados de la compilación y los informes de las pruebas.

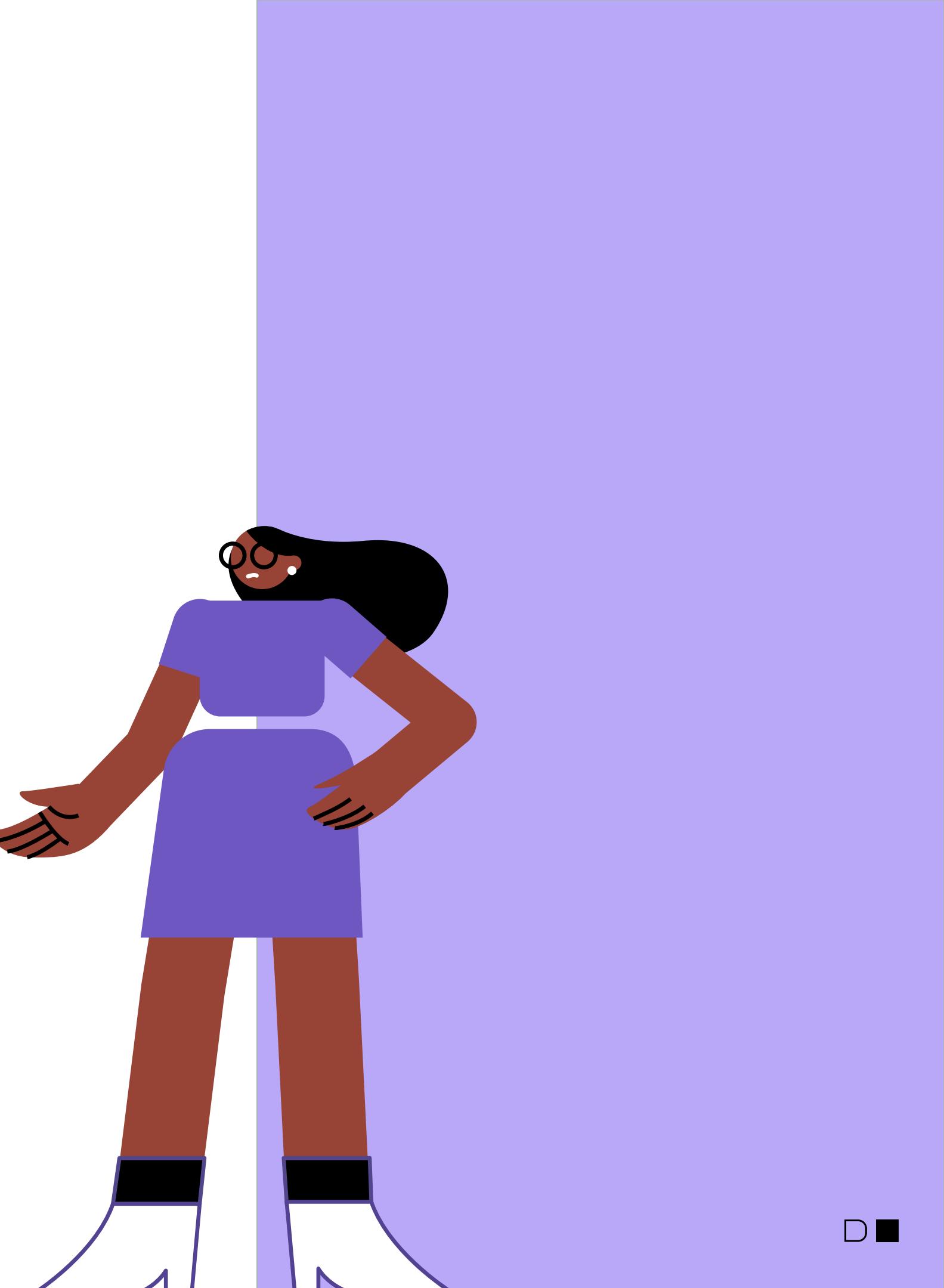


# Dependencias

Las **dependencias** son componentes que nuestro software necesitará para su correcta ejecución durante algún ciclo de vida.

Si descargamos un proyecto y lo implementamos en nuestro sistema, Maven obtendrá las dependencias del proyecto que son necesarias o bien desde el repositorio local, o desde un repositorio remoto.

Esto permite a los usuarios de Maven **reutilizar las dependencias entre proyectos** para garantizar que los problemas de compatibilidad con versiones anteriores se resuelvan adecuadamente.



# Anatomía de JUnit como dependencia

Al sumar el **JUnit** como dependencia se verá:

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.13.2</version>
    <scope>test</scope>
</dependency>
```

# Repositorio de dependencias

Un repositorio bastante popular de dependencias a nivel mundial cuando estamos usando Maven es:



[mvnrepository.com](https://mvnrepository.com)

# Before test

Para ejecutar cualquier script de prueba de Selenium, necesitarán un controlador de navegador y, para usarlo, necesitan establecer la ruta ejecutable del controlador de navegador explícitamente. Después de eso, es necesario instanciar la instancia del controlador y proceder con el caso de prueba.



```
@BeforeTest  
public void setUp() throws Exception {  
    WebDriverManager.chromedriver().setup();  
    driver=new ChromeDriver();  
}
```

# Before test

Al principio de la clase se pueden definir variables y rellenarlas con datos:

```
class FirstTestScriptUsingSeleniumGrid {  
  
    public String username = "Digital House";  
    public String accesskey = "CTD2022";  
    public static RemoteWebDriver driver = null;  
    public String gridURL = "@https://www.digitalhouse.com.br/";
```

# Entidades

Las entidades son **objetos** que se mantienen en memoria durante un intervalo de tiempo y luego se persisten en la base de datos. La clase **entidad** representa **una tabla de la base de datos** y el objeto **entidad** representa una **fila de la tabla**.

Por lo general, cumple con los requisitos de la empresa para proporcionarnos la capacidad de crear los datos necesarios para realizar la prueba.

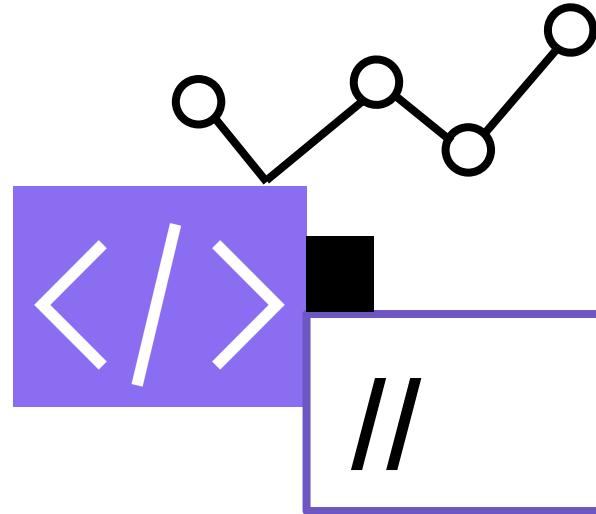
# Entidades

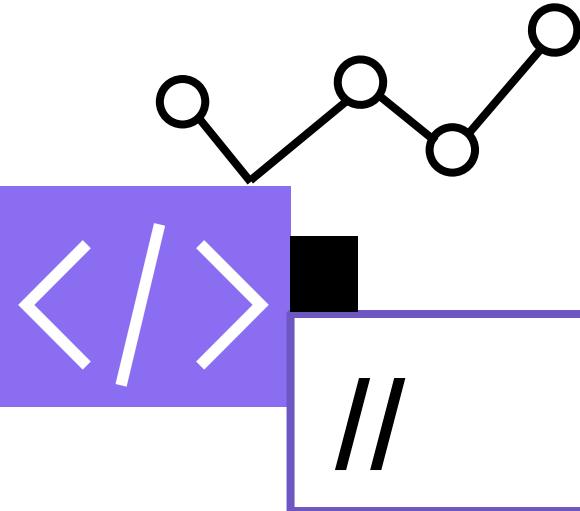
Imaginen por ejemplo que quieren probar un sistema de cuentas bancarias, para ello pueden crear la entidad **Cuenta**, indicando los atributos y métodos necesarios.

```
//Creating the bank account class

public class Account {
    private int NumberAccount;
    public int op;
    private String Name;
    private double AccountBalance;
    private double AccountLimit;

    public Account() {
        this.NumberAccount=0;
        this.Name=" ";
        this.AccountBalance=0;
        this.AccountLimit=0;
    }
    public int getNumberAccount() {
        return NumberAccount;
    }
    public void setNumberAccount(int NumberAccount) {
        this.NumberAccount = NumberAccount;
    }
    public String getName() {
        return Name;
    }
    public void setName(String Name) {
        this.Name = Name;
    }
    public double getAccountBalance() {
        return AccountBalance;
    }
    public void setAccountBalance(double AccountBalance) {
        this.AccountBalance = AccountBalance;
    }
    public double getAccountLimit() {
        return AccountLimit;
    }
    public void setAccountLimit(double AccountLimit) {
        this.AccountLimit = AccountLimit;
    }
    boolean withdrawal(double quantity) {
        if (this.AccountBalance<quantity)
            return false;
        else {
            this.AccountBalance = this.AccountBalance - quantity;
            return true;
        }
    }
    void deposit(double quantity) {
        this.AccountBalance = this.AccountBalance + quantity;
    }
    void fillinAccountData(String a, int b, float c, float l){
        this.Name = a;
        this.NumberAccount = b;
        this.AccountBalance = c;
        this.AccountLimit = l;
    }
    void showAccountInfo(){
        System.out.println("Name: " + this.getName());
        System.out.println("Number Account: " + this.getNumberAccount());
        System.out.println("Account Balance: " + this.getAccountBalance());
    }
    void showAccountBalance(){
        System.out.println("Account Balance : " + this.getAccountBalance());
    }
}
```



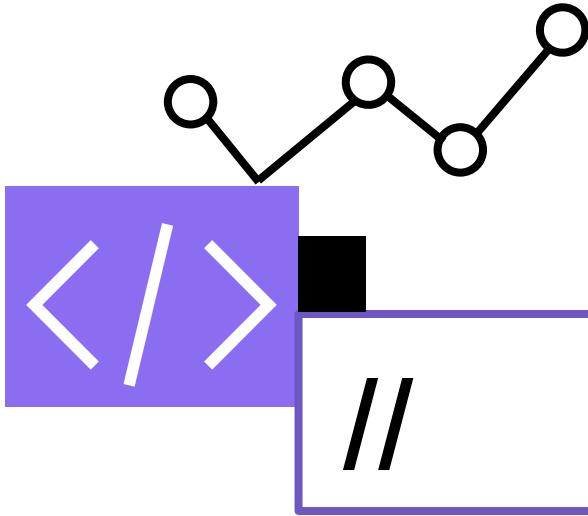


# Entidades

Por ejemplo, antes de probar la posibilidad de un retiro, tendría que crear una cuenta con los datos necesarios, para saber si la cuenta tiene suficiente saldo para liberar el retiro.

```
// creating the account
Account myAccount = new Account();

// filling in the account data
myAccount.fillinAccountData ("mari", 1020, 100.50, 1000.10)
```



# Entidades

Probando el retiro

```
● ● ●  
@Test  
public void tryWithdrawal() throws Exception {  
    //given  
    Account myAccount = new Account();  
    myAccount.fillinAccountData ("mari", 1020, 100.50, 1000.10)  
  
    //when  
  
    Boolean result = myAccount.withdrawal (2000);  
  
    //then  
    assertTrue("Don't allow to withdrawal" == result);  
}
```

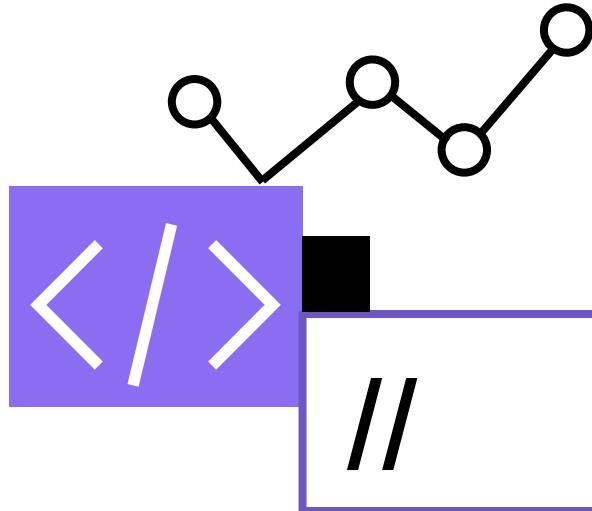


Independientemente de la forma en que decida crear y alimentar las variables y los objetos, lo más importante es tener en cuenta qué es lo que necesitamos ejecutar antes de nuestra prueba.

# After test

La anotación **@After** se utiliza para realizar tareas después de la ejecución de cada prueba. Por ejemplo:

- Enviar los resultados de las pruebas a un servicio de registro o supervisión.
- Cerrar el navegador.
- Crear informe después de ejecutar la prueba.
- Recoger capturas de pantalla.
- Las variables se pueden borrar o restablecer.
- Eliminar datos creados por la misma prueba.

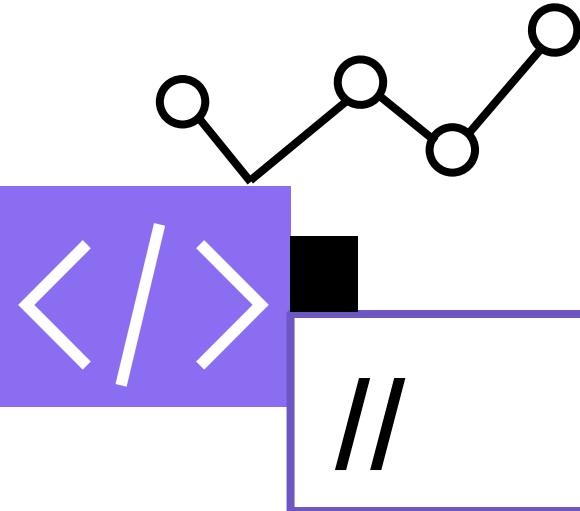


# After test



```
@AfterTest
    public void closeBrowser() {
        driver.close();
        System.out.println("The driver has been closed.");
        FirstTestScriptUsingSeleniumGrid.username = "";
        FirstTestScriptUsingSeleniumGrid.accesskey = "";

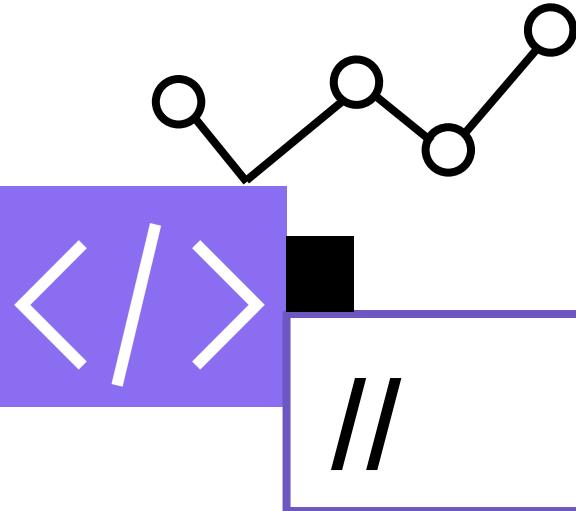
    }
```



# Informes y registros

En caso de que una prueba falle, **es vital tener capturas de pantalla a las que referirse**. Esto ayuda a explicar el fallo al desarrollador, que puede depurarlo al instante. Del mismo modo, desde el punto de vista de los informes, para proporcionar información a las partes interesadas, **es valioso compartir con ellas los informes**, para dar visibilidad a la calidad del producto.

Veremos ejemplos de pruebas utilizando **Extent Report** para generar informes, registros y capturar pantallas.



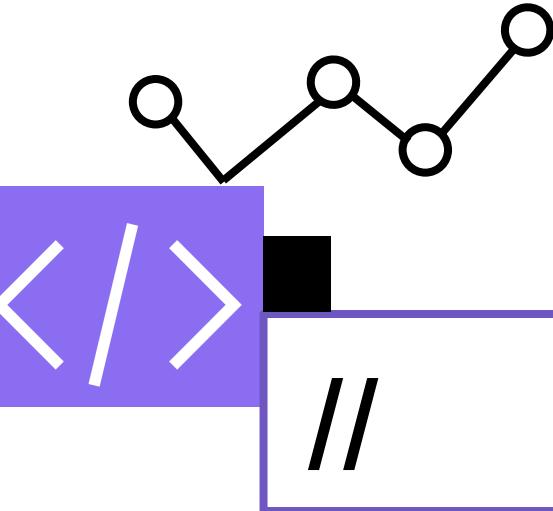
# Informes y registros

Intenten capturar pantallas con el siguiente código:

```
● ● ●

test.log(LogStatus.FAIL,test.addScreenCapture(capture(driver))+ "Test Failed");

public static String capture(WebDriver driver) throws IOException {
    File scrFile = ((TakesScreenshot) driver).getScreenshotAs(OutputType.FILE);
    File Dest = new File("src/../../BStackImages/" + System.currentTimeMillis()
    + ".png");
    String errflpath = Dest.getAbsolutePath();
    FileUtils.copyFile(scrFile, Dest);
    return errflpath;
}
```

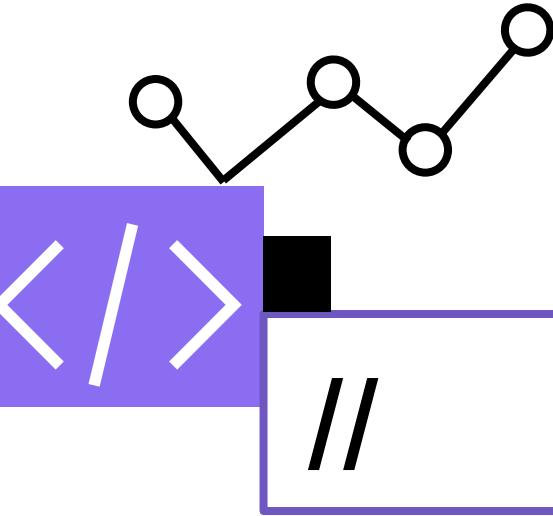


# Informes y registros

**getScreenShotAs()**: Captura la pantalla de la instancia actual de WebDriver y la almacena en diferentes formas de salida.

```
File scrFile = ((TakesScreenshot) driver).getScreenshotAs(OutputType.FILE);
```

Este método devuelve un objeto de archivo que se almacenará en una variable de archivo.

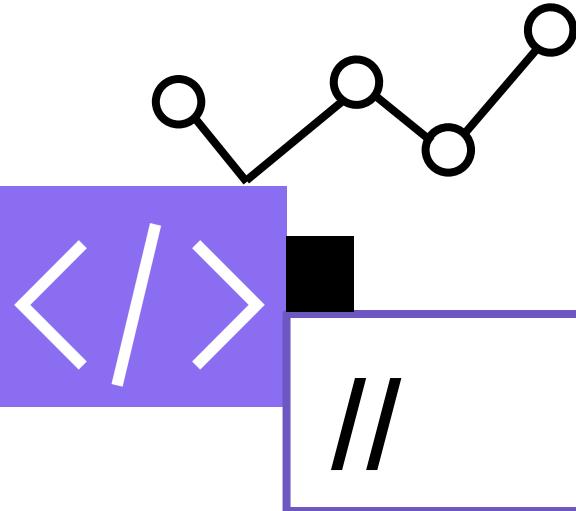


# Informes y registros

Esta sentencia crea una carpeta llamada '**BStackImages**' dentro de la carpeta 'src' y almacena el nombre del archivo como la hora actual del sistema.



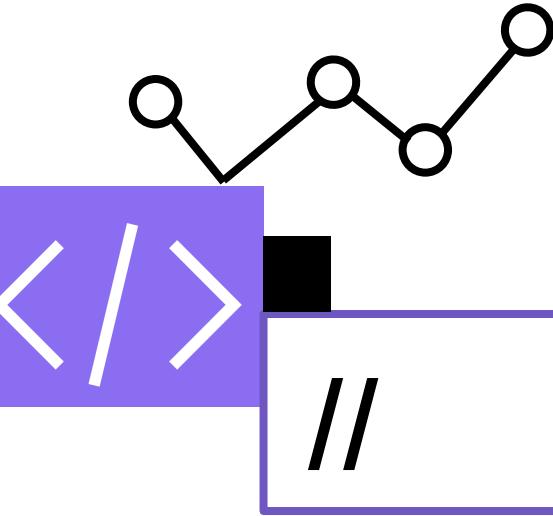
```
File Dest = new File("src/../../BStackImages/" + System.currentTimeMillis() + ".png");
```



# Informes y registros

Estas instrucciones copian todas las imágenes de error en la carpeta de destino.

```
String errflpath = Dest.getAbsolutePath();
FileUtils.copyFile(scrFile, Dest);
return errflpath;
```



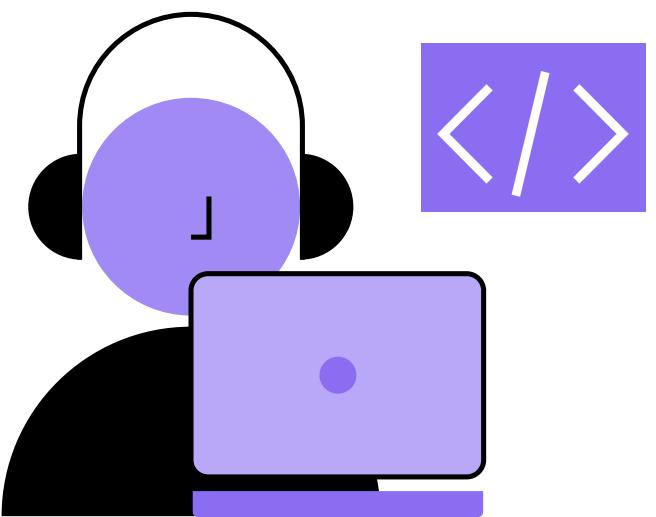
# Informes y registros

Usar el método log porque utiliza el método addScreenCapture de la clase Extent Test para tomar una captura de pantalla y añadirla al Extend Report.



```
test.log(LogStatus.FAIL,test.addScreenCapture(capture(driver))+ "Test Failed");
```

## Ejemplos de script completo:



```
import org.apache.commons.io.FileUtils;
import org.openqa.selenium.By;
import org.openqa.selenium.OutputType;
import org.openqa.selenium.TakesScreenshot;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.interactions.Actions;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;
import org.testng.Reporter;
import org.junit.Assert;
import org.junit.BeforeClass;
import org.junit.Test;
import org.junit.AfterClass;

public class BrowseWikipedia {
    public static WebDriver driver;
    public Boolean shouldTheTestPass = false;

    @BeforeTest
    public static void setUp() throws Exception{
        driver = new FirefoxDriver();
    }

    @Test
    public void step01_HomePage() throws Exception{
        listEnvironmentVariables();

        driver.get("http://www.wikipedia.org/");
        String i = driver.getCurrentUrl();
        System.out.println(i);
        BrowseWikipedia me = new BrowseWikipedia();
        try{
            me.takeScreenShot("Step-1.png",driver);
        } catch(Exception e){
            System.out.println("Problem, "+e.toString());
        }
    }

    @Test
    public void step02_MainPage() throws Exception{
        driver.findElement(By.linkText("English")).click();

        BrowseWikipedia me = new BrowseWikipedia();
        try{
            me.takeScreenShot("Step-2.png",driver);
        } catch(Exception e){
            System.out.println("Problem, "+e.toString());
        }
    }

    @AfterTest
    public static void tearDown() throws Exception{
        driver.quit();
    }
}
```

## Ejemplos de script completo:



```
package LambdaTest;

import io.github.bonigarcia.wdm.WebDriverManager;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.annotations.AfterTest;
import org.testng.annotations.BeforeTest;
import org.testng.annotations.Listeners;
import org.testng.annotations.Test;

@Listeners({util.Listener.class})
class FirstTestScriptUsingWebDriver {
    public static WebDriver driver = null;
    @BeforeTest
    public void setUp() throws Exception {
        WebDriverManager.chromedriver().setup();
        driver=new ChromeDriver();
    }
    @Test
    public void firstTestCase() {
        try {
            System.out.println("Logging into Lambda Test Sign Up Page");
            driver.get("https://accounts.lambdatest.com/register");
            WebElement pageHeader= driver.findElement(By.xpath("//a[text()='Sign In']"));
            pageHeader.click();
            System.out.println("Clicked on the Sign In Button.");
        } catch (Exception e) {
        }
    }
    @AfterTest
    public void closeBrowser() {
        driver.close();
        System.out.println("The driver has been closed.");
    }
}
```

# Uso de anotaciones

Las anotaciones de prueba son básicamente metadatos utilizados para determinar qué acción debe realizar un método. Permiten a los desarrolladores organizar, agrupar y mantener los casos de prueba. Selenium permite la integración con JUnit y TestNG...

## JUnit

Publicado por primera vez en 2002, el marco JUnit está diseñado para escribir casos de prueba para Java. Cuando las aplicaciones se construyen a escala, Java es el lenguaje de programación preferido. La investigación sobre los lenguajes de programación utilizados para construir sitios web de alto tráfico revela que están basados principalmente en Java.

Las anotaciones de JUnit son una de las principales razones de la popularidad del framework, estas son fundamentales y permiten al marco de trabajo identificar una tarea específica que debe ser ejecutada.

Se pueden combinar muchas pruebas en una sola clase de prueba utilizando este marco. Todos los casos de prueba pueden necesitar un procedimiento de inicialización común que se realice antes de la prueba en sí. Es posible que sea necesario realizar alguna limpieza —cerrar archivos, liberar recursos, etc.— tras la ejecución de una prueba. JUnit maneja estos "requisitos" haciendo uso de anotaciones. De este modo, el framework sabe distinguir explícitamente entre las tareas de inicialización, las pruebas propiamente dichas y las tareas posteriores a la ejecución.

**@BeforeClass**

La anotación **@BeforeClass** se ejecuta sólo una vez y específicamente antes de que se ejecute cualquier otra cosa en la clase de prueba. Esto es extremadamente útil para las pruebas de JUnit Selenium porque el programa debe abrir un navegador antes de ejecutar cualquiera de las pruebas de Selenium. Así, un método que inicializa el Selenium Webdriver y abre una conexión a la base de datos puede ser anotado con la anotación **@BeforeClass**.

• • •

```
@RunWith (JUnit4.class)
public class
BeforeClassAndAfterClassAnnotationsUnitTest {
//...
@BeforeClass
public static void setup () {
LOG.info("startup - creating DB connection");
}}
```

>

### @AfterClass

La anotación **@AfterClass**, como su nombre podría sugerir, es la anotación que se utiliza para ejecutar las tareas después de que todas las pruebas hayan terminado. Por ejemplo, **@AfterClass** se utilizará para cerrar una conexión de base de datos y liberar ese recurso una vez que se hayan ejecutado todas las pruebas dentro de la clase de prueba de JUnit Selenium.

```
...  
  
@AfterClass  
public static void tearDown () {  
    LOG.info("closing DB connection");  
}
```

### @After

Del mismo modo, la anotación **@After** se utiliza para realizar tareas después de la ejecución de cada prueba. Después de cada prueba, puede ser necesario enviar los resultados de la misma a un servicio de registro o de supervisión. Tenga en cuenta que también hemos añadido otro método anotado con **@After** para limpiar la lista después de la ejecución de cada prueba.

```
...  
  
@After  
public void teardown() {  
    LOG.info("teardown");  
    list.clear();  
}
```

Consideren el siguiente escenario. Se crea una aplicación web para garantizar que no se produzca el almacenamiento en caché debido a los datos altamente dinámicos. Así, cada vez que se carga la página, la caché debe estar vacía. Para automatizar la prueba de este escenario, es necesario ejecutar la misma secuencia varias veces.

Con las anotaciones de JUnit 5, se ha añadido la anotación **@RepeatedTest**. Esto se puede utilizar para ejecutar una prueba determinada cualquier número de veces. La creación de una prueba repetida es sencilla: basta con añadir la anotación **@RepeatedTest** en la parte superior del método de prueba.

### @Before

Estas dos anotaciones pretenden ayudar a los desarrolladores de automatización a escribir código que realice tareas antes de que se ejecute cada prueba. Al escribir una clase de prueba de Selenium, puede haber un paso obligatorio que inicie la prueba desde una página web específica. Esto es útil cuando queremos ejecutar algún código común **antes de ejecutar una prueba**. Vamos a inicializar una lista y añadir algunos valores.

```
...  
  
@RunWith(JUnit4.class)  
public class BeforeAndAfterAnnotationUnitTest {  
  
    // ...  
  
    private void init() {  
        LOG.info("startup");  
        list = new ArrayList <> (Arrays.asList("test1",  
            "test2"));  
    }  
  
}
```

La anotación **@Test** se utiliza para identificar el caso de prueba real. Esto es necesario porque JUnit permite agrupar varias pruebas en una sola clase de prueba. El método de prueba es donde se hacen las afirmaciones y se determina el resultado.

```
...  
  
@RepeatedTest(3)  
void repeatedTest(TestInfo testInfo) {  
    System.out.println("Executing repeated test");  
    assertEquals (2, Math.addExact(1,1), "1 + 1  
should equal 2");  
}
```

Tengan en cuenta que en lugar de la anotación estándar **@Test**, estamos utilizando **@RepeatedTest** para nuestra prueba unitaria. La prueba anterior se ejecutará tres veces, como si se hubiera escrito la misma prueba tres veces.



# Atributos de las anotaciones de JUnit y TestNG

## JUnit

Algunas anotaciones de JUnit aceptan parámetros llamados **atributos** que se utilizan para proporcionar más detalles a la prueba que se está ejecutando. Selenium requiere estos parámetros para aplicar restricciones como el límite de tiempo, el número de veces que debe ejecutarse una prueba, etc. Veamos un ejemplo y exploremos algunos de estos atributos de anotación.

En el siguiente fragmento de código, se utilizan dos anotaciones **@Test** y **@RepeatedTest**. A cada uno de ellos se le ha asignado un atributo. La primera anotación **@Test** se ha establecido con el atributo **timeout** (en milisegundos) que indica a Selenium que establezca el tiempo de espera en 5 segundos. Del mismo modo, la segunda anotación **@RepeatedTest** recibe un número entero que le dice a JUnit que ejecute la prueba llamada "test\_search" seis veces.

Pasar atributos para algunas anotaciones es opcional, mientras que para otras es obligatorio. En este caso, el atributo de anotación **@Test** es opcional, en tanto que el atributo **@RepeatedTest** es obligatorio.

```
@Test(timeout=5000)
@RepeatedTest(6)
public void test_search()
{
    // code to search something using the search_bar
    driver.findElement(By.id("search_bar")).sendKeys("Browserstack automation testing");
    driver.findElement(By.name("search_btn")).click();

    String first_result = driver.findElementByXPath("/html/body/result").getText();

    assertEquals(first_result,"Testing and annotations");
}
```

## TestNG

TestNG es un marco de automatización de pruebas en el que NG significa "**Next Generation**". TestNG se inspira en JUnit, que utiliza las anotaciones (@). TestNG supera las desventajas de JUnit y está diseñado para facilitar las pruebas de extremo a extremo. Las anotaciones en TestNG son líneas de código que pueden controlar cómo se ejecutarán los métodos que tienen debajo. Siempre van precedidos del símbolo @. A continuación se muestra un ejemplo de un TestNG muy temprano y rápido.

```

@Test(priority = 0) // example of annotations
public void goToHomepage(){
    driver.get(baseUrl);
    Assert.assertEquals(driver.getTitle(), "Welcome: Mercury Tours");
}

@Test(priority = 1) // example of annotations
public void logout(){
    driver.findElement(By.LinkText("SIGN-OFF")).click();
    Assert.assertEquals("Sign-on: Mercury Tours", driver.getTitle());
}

```

El ejemplo anterior simplemente dice que el método goToHomepage() debe ejecutarse primero antes de logout() porque tiene un número de prioridad más bajo.

## **Resumen de anotaciones de TestNG**

**@BeforeSuite:** el método anotado se ejecutará antes de que se ejecuten todas las pruebas de este conjunto.

**@AfterSuite:** el método anotado se ejecutará después de que se ejecuten todas las pruebas de este conjunto.

**@BeforeTest:** el método anotado se ejecutará antes de que se ejecute cualquier método de prueba perteneciente a las clases dentro de la etiqueta.

**@AfterTest:** el método anotado se ejecutará después de la ejecución de todos los métodos de prueba pertenecientes a las clases dentro de la etiqueta.

**@BeforeGroups:** la lista de grupos que este método de configuración ejecutará antes. Se garantiza que este método se ejecute justo antes de que se invoque el primer método de prueba que pertenezca a cualquiera de estos grupos.

**@AfterGroups:** la lista de grupos después de los cuales se ejecutará este método de configuración. Se garantiza que este método se ejecute justo después de que se invoque el último método de prueba que pertenezca a cualquiera de estos grupos.

**@BeforeClass:** el método anotado se ejecutará antes de que se invoque el primer método de prueba de la clase actual.

**@AfterClass:** el método anotado se ejecutará después de que se ejecuten todos los métodos de prueba de la clase actual.

**@BeforeMethod:** el método anotado se ejecutará antes de cada método de prueba.

**@AfterMethod:** el método anotado se ejecutará después de cada método de prueba.

**@Test:** el método anotado forma parte de un caso de prueba.

Las anotaciones se utilizan para describir un lote de código insertado en el programa o la lógica de negocio utilizada para controlar el flujo de los métodos en el script de prueba. Hacen que los scripts de prueba de Selenium sean más manejables, sofisticados y eficaces. Su uso es extremadamente útil para los probadores y les facilita mucho la vida.

## ¿Cómo esperar por un elemento?

Muchas veces, cuando automatizamos pruebas dependemos de la velocidad de respuesta de la página web que estamos probando. A veces, esta web va más lenta de lo que esperamos y eso hace que nuestras pruebas fallen, ya que intentan interactuar con un elemento que todavía no está disponible o visible. Esta situación se da muy a menudo en proyectos de automatización y es por ello que Selenium nos provee de herramientas para solucionarlo.

Lo primero que podemos hacer es implementar una **espera implícita**, es decir, esperar por un tiempo determinado hasta que el elemento en cuestión esté presente y disponible. Esto, si bien puede solucionar nuestro problema, no es una buena práctica porque:

- Nos hace esperar por un tiempo fijo, independientemente si el objeto aparece o no.
- La espera fija no nos asegura que el objeto aparezca.

Para mejorar esto, lo que Selenium nos provee son **esperas explícitas** en donde podemos esperar por un elemento en un rango de tiempo de N segundos tal como se define en la clase «WebDriverWait» o hasta que se cumplan las condiciones de espera. Selenium provee las siguientes condiciones de espera:

- alertIsPresent()
- elementSelectionStateToBe()
- elementToBeClickable()
- elementToBeSelected()
- frameToBeAvailableAndSwitchToIt()
- invisibilityOfTheElementLocated()
- invisibilityOfElementWithText()
- presenceOfAllElementsLocatedBy()
- presenceOfElementLocated()

- `textToBePresentInElement()`
- `textToBePresentInElementLocated()`
- `textToBePresentInElementValue()`
- `titleIs()`
- `titleContains()`
- `visibilityOf()`
- `visibilityOfAllElements()`
- `visibilityOfAllElementsLocatedBy()`
- `visibilityOfElementLocated()`

De esta forma, solucionamos el uso de `sleep()` que se venía usando hasta el momento.

## Resumiendo..

Algunas anotaciones son necesarias para determinar el orden de ejecución de los bloques de código. Así conocemos las anotaciones:

- **@BeforeTest:** el método anotado se ejecutará antes de que se ejecute cualquier método de prueba perteneciente a las clases dentro de la etiqueta.
- **@AfterTest:** el método anotado se ejecutará después de la ejecución de todos los métodos de prueba pertenecientes a las clases dentro de la etiqueta.

Cabe destacar que las versiones del framework JUnit sufren cambios, por lo que es importante consultar la documentación de la versión utilizada.

Además, existen dos grandes formas de esperar por un elemento en un proyecto de automatización: de forma implícita y de forma explícita.

# Revisión: primera parte

# Índice

- 01** [Patrones de diseño para la automatización de pruebas](#)
- 02** [Ubicación de los elementos](#)
- 03** [Before y After test](#)
- 04** [Uso de anotaciones](#)



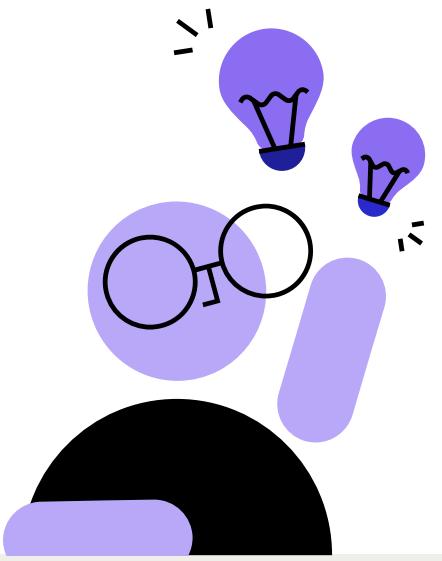
01

# Patrones de diseño para la automatización de pruebas

**La automatización de pruebas** mediante herramientas como **Selenium** y otras, son esencialmente actividades de desarrollo.

Se necesitan buenos conocimientos de programación y patrones de diseño para implementar pruebas robustas, estables, eficientes, rápidas y sostenibles.





# Patrones de diseño

## Modelo Page Object

El modelo **Page Object**, también conocido como POM, es un patrón de diseño que **crea un repositorio de objetos para almacenar todos los elementos de la web**. Es útil para reducir la duplicación de código y mejorar el mantenimiento de los casos de prueba. En el Modelo Page Object, consideren cada página web de una aplicación como un archivo de clase. Cada archivo de clase contendrá sólo los elementos correspondientes de la página web. Con estos elementos, los probadores pueden realizar operaciones en el sitio en prueba.

## Screenplay

Es un enfoque centrado en el usuario para escribir pruebas de aceptación automatizadas de alta calidad. Orienta el uso eficaz de las capas de abstracción y fomenta los buenos hábitos de prueba e ingeniería del software. Se compone de cinco elementos, cinco tipos de bloques de construcción que le sirven para diseñar cualquier prueba de aceptación funcional que necesite, por sofisticada o sencilla que sea. Los elementos clave son: actores, habilidades, interacciones, preguntas y tareas.

# Marcos de trabajo para la automatización de pruebas

Se trata de un conjunto de directrices o reglas utilizadas para **crear y diseñar casos de prueba**. Un framework compuesto por una combinación de prácticas y herramientas diseñadas para ayudar a los profesionales del control de calidad a realizar pruebas de forma más eficaz. Estas directrices pueden incluir normas de codificación, métodos de manipulación de datos de prueba, repositorios de objetos, procesos de almacenamiento de resultados de pruebas o información sobre cómo acceder a recursos externos.

Hay seis tipos comunes de frameworks de automatización de pruebas, y a la hora de crear un plan de pruebas es importante elegir el adecuado para usted.

02

## Ubicación de los elementos

# Ubicación de los elementos

**Todo lo que se ve en la página es un elemento.** Todos los campos, enlaces, imágenes, textos y muchas cosas que no se ven son elementos, aunque puede haber elementos en una página que no provengan de la fuente HTML.

- **Localizador de elementos** - Es un método para localizar un elemento en una página. Hay muchos tipos diferentes de localizadores, decidir cuál usar depende de muchos factores diferentes y no hay una respuesta correcta más que la que funcione para ustedes. En muchos casos, se pueden utilizar varios localizadores diferentes para realizar la misma tarea.

# Ubicación de los elementos

- **Encontrar e inspeccionar el elemento** - Necesitamos saber con qué estamos trabajando, así que el siguiente paso es siempre mirar el elemento en la página para ver qué propiedades tiene, y que podemos usar para identificarlo. El navegador (o un plugin) te ayudará a hacerlo.



The screenshot shows the Chrome DevTools Elements tab with the following DOM structure:

```
<h1>...</h1>
<form name="loginform" id="loginform" action="http://ec2-54-82-87-239.compute-1.amazonaws.com/wordpress/wp-login.php?_wpnonce=4f333a3a33&redirect_to=http%3A%2F%2Fec2-54-82-87-239.compute-1.amazonaws.com%2Fwordpress%2F&reauth=1" method="post" style="margin-bottom: 15px;">
  <p>...</p>
  <p>
    <label for="user_pass">
      "Password"
      <br>
      <input type="password" name="pwd" id="user_pass" class="input" value size="20">
    </label>
  </p>
  <p class="forgetmenot">...</p>
  <p class="submit">...</p>
</form>
<p id="nav">...</p>
<script type="text/javascript">...</script>
<p id="backtoblog">...</p>
```

The `<input type="password" name="pwd" id="user_pass" class="input" value size="20">` line is highlighted with a blue selection bar, indicating it is the currently selected element.

# Ubicación de los elementos

- **Elian un localizador** - Después de inspeccionar el elemento, es el momento de intentar localizarlo. Entre las diversas estrategias siguen algunos ejemplos como:

El atributo **id**: el campo de la contraseña tiene un atributo ID (id="user\_pass"), que normalmente sería la primera opción. Pero como ya hemos determinado que el ID cambia con cada iteración, sabemos que esto no funcionará en este caso.

El atributo **name**: este elemento es un elemento de entrada a un formulario y tiene un atributo name (name="pwd"). Un nombre de campo suele ser una buena segunda opción, después del id del elemento. Pruebe un localizador de nombres de campo con el nombre "pwd".

El atributo **class**: el elemento tiene una clase (atributo) CSS. Si este es el primer (o único) elemento de la página que utiliza esta clase, entonces el elemento puede ser localizado en el localizador de clases CSS con la clase "input" ..

# Ubicación de los elementos

- **XPath:** Un XPath siempre está disponible para un elemento, de hecho, normalmente hay múltiples XPaths válidos para un elemento. XPath es un recurso potente y flexible. Sabemos que el navegador puede proporcionar un XPath, sin embargo, como este elemento tiene un id, el navegador generará un XPath basado en el id que ya sabemos que no podemos utilizar en este caso. Podemos ver que el formulario padre también tiene un id, si ese id no cambia, un XPath desde ese elemento al campo contraseña debería funcionar.

```
//*[@id="loginform"]/p[2]/label/input
```

Este XPath sigue utilizando un ID, pero el ID del formulario en lugar del campo de la contraseña. Si el ID del formulario también cambia con cada vista de la página, esto tampoco funcionará y habrá que considerar otro tipo de localización.

# Ubicación de los elementos

- **Seletor CSS:** Muchos elementos pueden seleccionarse fácilmente utilizando selectores CSS. En el ejemplo anterior, el elemento tiene una clase CSS de entrada y es probablemente el único campo de entrada en la página con el atributo `type='password'`, por lo que este selector CSS podría funcionar: `[type='password'].input`
- **Link Text:** Los links se pueden localizar haciendo coincidir el texto del enlace con el parámetro proporcionado.
- **Text:** Los elementos de uso común que tienen una etiqueta o un texto visible pueden localizarse a menudo mediante este sencillo localizador.

03

## Before y After test

# Métodos para ejecutar antes y después de cada test

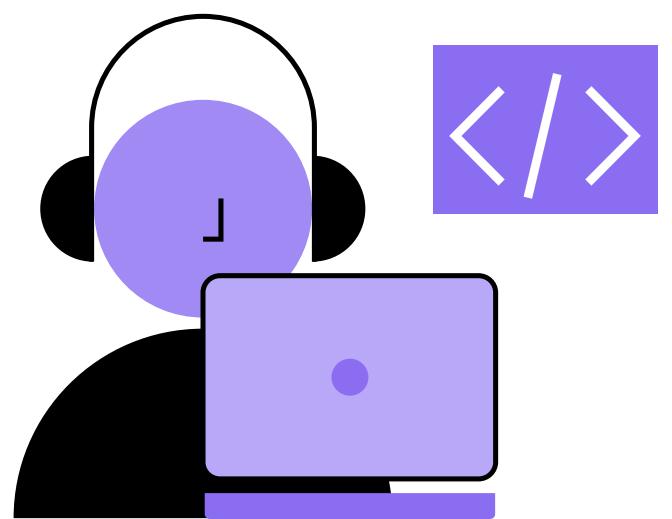
El método **@Before** se ejecuta antes de la ejecución de una prueba.

El método **@After**, en cambio, se ejecuta después de cada prueba. Esto se puede utilizar para configurar repetidamente una prueba, y borrarla después de cada ejecución.

Por lo tanto, las pruebas son independientes y el código de configuración no se copia en el método de prueba y ambos deben ser public void.



## Ejemplo



```
import static org.junit.Assert.assertEquals;  
  
import java.util.ArrayList;  
import java.util.List;  
  
import org.junit.After;  
import org.junit.Before;  
import org.junit.Test;  
  
public class DemoTest{  
    private List<Integer> list;  
  
    @Before  
    public void setUp(){  
        list = new ArrayList<>();  
        list.add(3);  
        list.add(1);  
        list.add(4);  
        list.add(1);  
        list.add(5);  
        list.add(8);  
    }  
  
    @After  
    public void tearDown(){  
        list.clear();  
    }  
  
    @Test  
    public void shouldBeOkAfterTestData(){  
        list.remove(0); // Remove first element of list  
        assertEquals(5, list.size()); // Size is down to five  
    }  
  
    @Test  
    public void shouldBeIndependentOfOtherTests(){  
        assertEquals(6, list.size());  
    }  
}
```

# Tareas comunes antes de la ejecución (utilicen `@BeforeTest`):

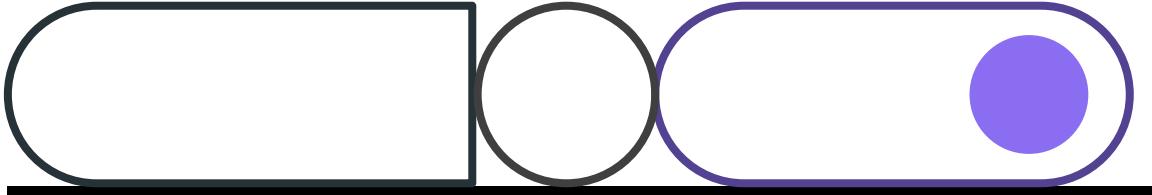
- Abran el navegador.
- Establecer una página web/sitio web/URL específica para iniciar.
- Limpieza de la caché del navegador.

# Tareas comunes después de la ejecución (utilice @AfterTest):

- Envíen los resultados de las pruebas a un servicio de registro o supervisión.
- Cierre el navegador.
- Crean un informe después de ejecutar la prueba.
- Recojan capturas de pantalla.
- Las variables se pueden borrar o restablecer.

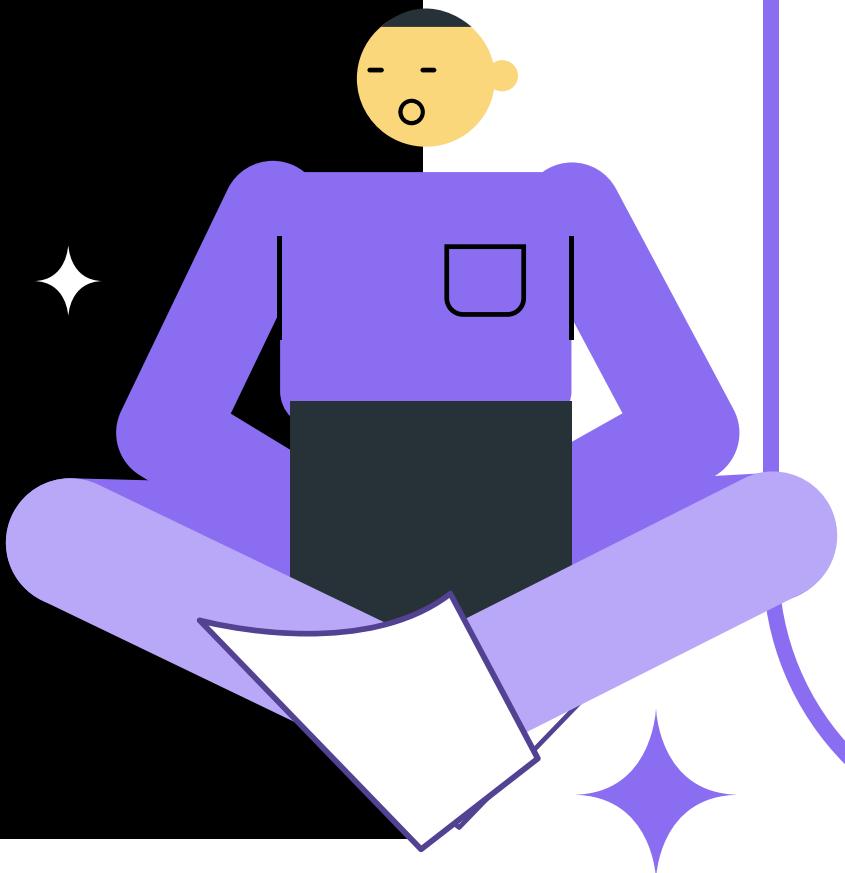
04

# Uso de anotaciones



Las anotaciones se utilizan para **controlar el siguiente método a ejecutar** en el script de prueba. Se colocan antes de cada método en el código de prueba. Si algún método no está prefijado con anotaciones, será ignorado y no se ejecutará como parte del código de prueba.

Para definirlos, basta con anotar los métodos con '**@Test**'.



# Anotaciones

A continuación se muestra la lista de anotaciones que lo soportan en Selenium:

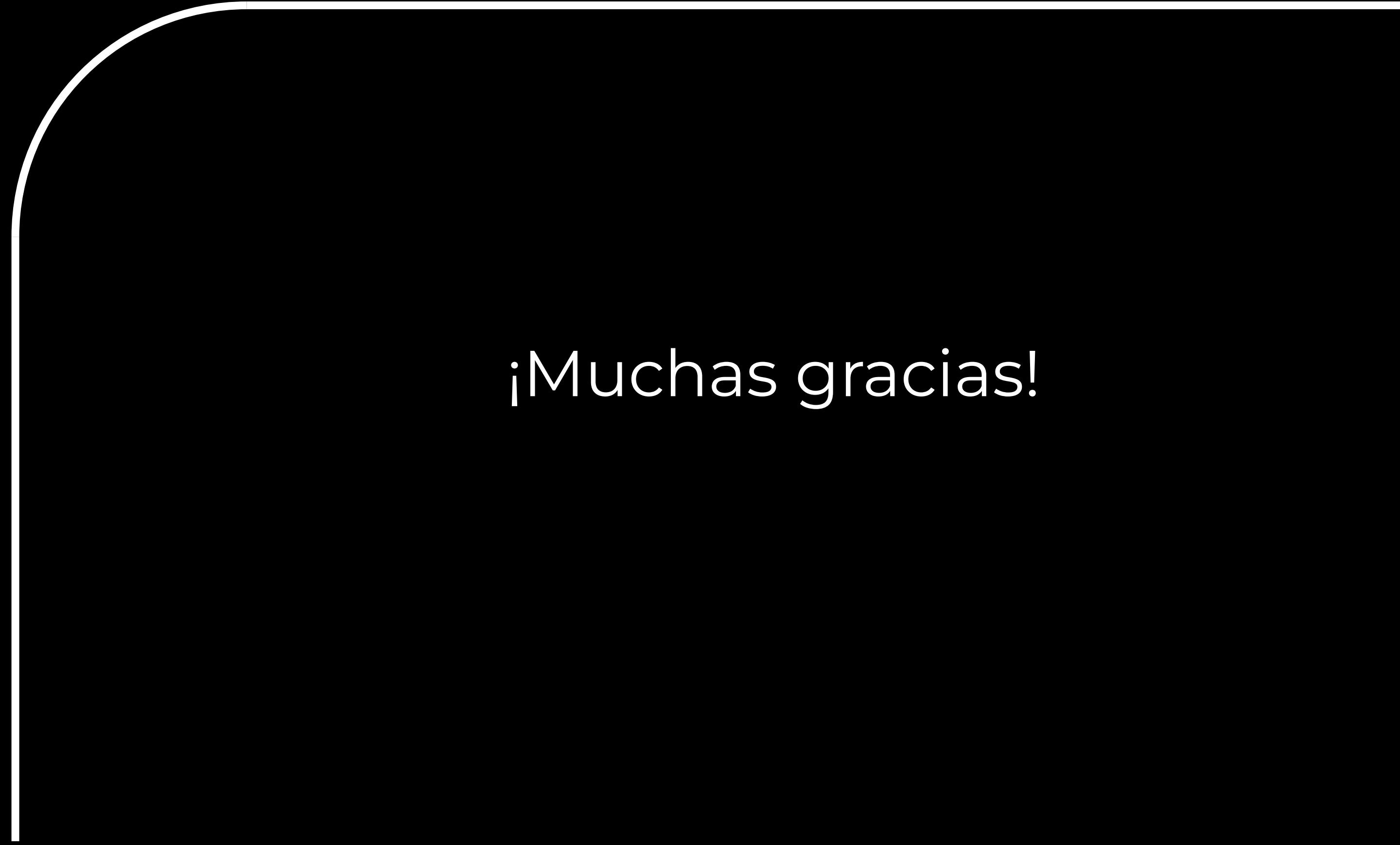
- **@BeforeMethod:** Se ejecutará antes de cada método anotado @test.
- **@AfterMethod:** Se ejecutará después de cada método anotado en @test.
- **@BeforeClass:** Se ejecutará antes de la primera ejecución del método @Test. Se ejecutará sólo una vez en todo el caso de prueba.
- **@AfterClass:** Se ejecutará después de que se hayan ejecutado todos los métodos de prueba de la clase actual.
- **@BeforeTest:** Se ejecutará antes del primer método @Test anotado. Se puede ejecutar varias veces antes del caso de prueba.

# Anotaciones

A continuación se muestra la lista de anotaciones que lo soportan en Selenium:

- **@AfterTest:** Un método con esta anotación se ejecutará cuando todos los métodos anotados a @Test hayan terminado de ejecutar esas clases dentro de la etiqueta <test> en el archivo TestNG.xml.
- **@BeforeSuite:** Se ejecutará sólo una vez, antes de que se ejecuten todas las pruebas de la suite.
- **@AfterSuite:** Un método con esta anotación se ejecutará una vez que se hayan completado todas las pruebas de la suite.
- **@BeforeGroups:** Este método se ejecutará antes de la primera prueba de ese grupo específico.
- **@AfterGroups:** Este método se ejecutará después de que todos los métodos de prueba de ese grupo hayan terminado su ejecución.





¡Muchas gracias!

## Armado de suites

Para empezar, recordemos la definición formal de un conjunto de pruebas y cuáles son los más comunes.

Test Suite es el conjunto de casos de prueba o scripts de prueba para ser ejecutados en un ciclo de ejecución específico” (ISTQB- Foundation Level)

Aunque cada equipo puede agrupar los casos de prueba según lo convenido, existen 2 tipos de test suite genéricas que están presentes en la mayoría de los proyectos. Veamos cuáles son..

The infographic is titled "Test suite genéricas" and compares two types of generic test suites: SMOKE and REGRESSION.

**SMOKE Test suite**

- Icon: A circular icon containing a stylized cloud or smoke.
- Description: Combina una serie de SMOKE tests en un pequeño conjunto de pruebas para que cuando la aplicación esté lista para entregarse a entornos de prueba o preproducción, se pueda saber en cuestión de minutos si las principales funcionalidades de la aplicación, sin entrar en detalle, están andando según lo esperado. Es decir, si la última versión de su aplicación está lista para que se comience a utilizar en un ciclo / sprint para un mayor control de calidad y revisión.
- Description: Las pruebas SMOKE también se utilizan en situaciones de emergencia o cuando los equipos quieren lanzar un parche a sus aplicaciones de producción. Aunque esta última práctica no se recomienda.
- Description: El gran beneficio de ejecutar regularmente un smoke test suite es el poder saber en forma temprana si se va a seguir trabajando con el release actual o se descarta.

**REGRESSION Test Suite**

- Icon: A circular icon containing a gear with a clock symbol.
- Description: Combina una serie de pruebas de regresión en un conjunto de pruebas, las cuales consisten en volver a ejecutar pruebas funcionales y no funcionales para garantizar que el software desarrollado y probado previamente siga funcionando después del cambio.
- Description: Se sigue un enfoque de prueba total, brinda la certeza de que los cambios realizados en el software no han afectado la funcionalidad existente, la cual debe permanecer inalterada.
- Description: Los cambios que pueden requerir pruebas de regresión incluyen el entorno de software, la corrección de errores, los cambios de configuración e incluso la sustitución de componentes electrónicos.

**Nota:** Es frecuente que la automatización de pruebas esté ligada a las pruebas de regresión debido a que estas son las que más tardan en ejecutarse: son pruebas estables y tienden a crecer con cada defecto encontrado. Sin embargo, contar con una suite de smoke test automatizada que se ejecuta rápidamente va a ahorrar tiempos en el proyecto y traer el feedback temprano de cada release.

# Planes de ejecución de prueba

Teniendo en cuenta lo antes mencionado, ahora podemos basarnos en estas dos suite para agrupar los tests automáticos y, de esta manera, contar con dos planes regulares de ejecución de prueba:

- Plan de ejecución de prueba de Smoke test suite
- Plan de ejecución de prueba de Regression test suite

Desde nuestro framework de trabajo que consta de **Java + Maven + Selenium** podemos utilizar la librería juniper de JUnit, la siguiente annotation nos permite agrupar los test:

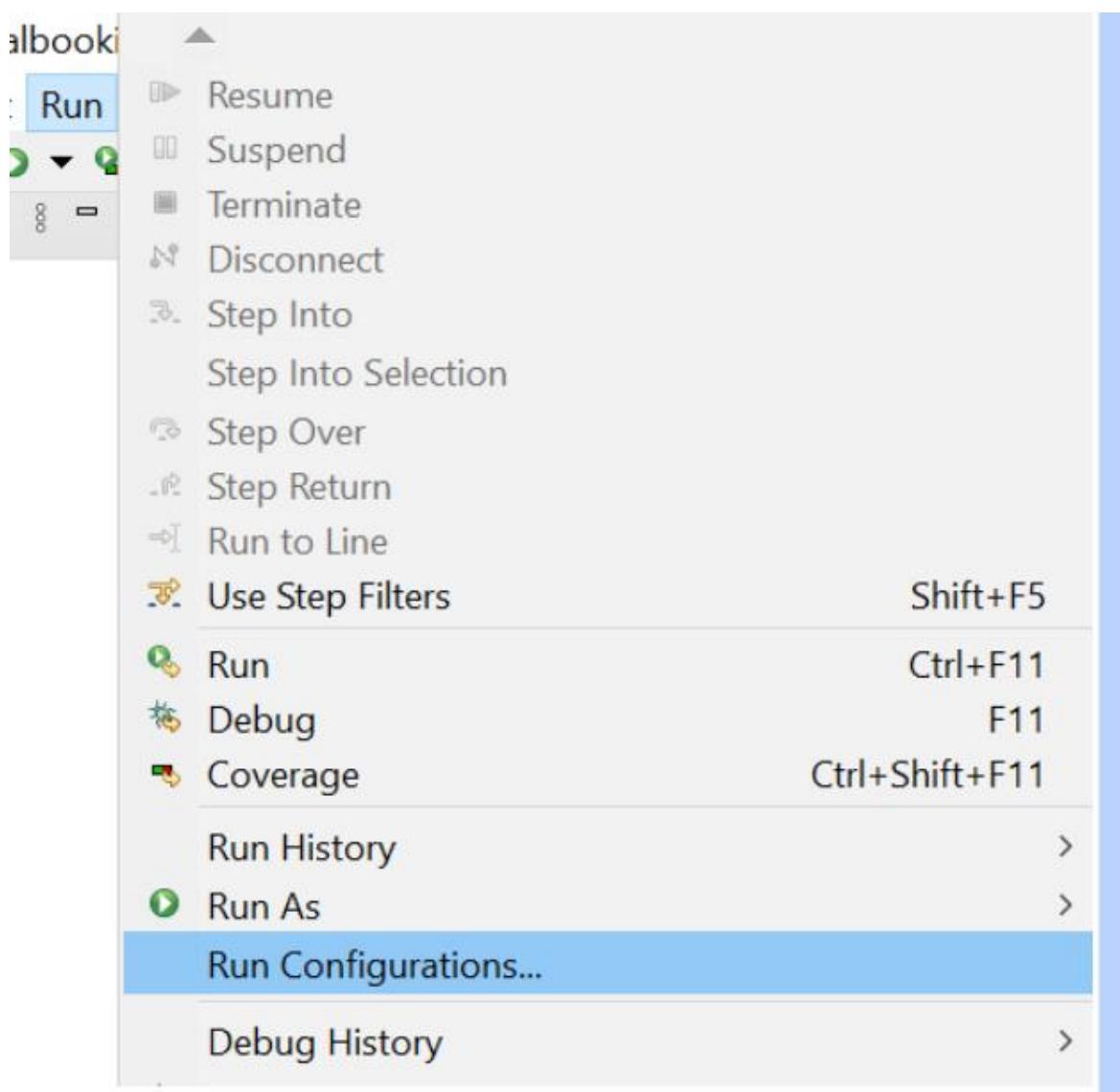
**@Tag:** Esta annotation deberá agregarse previo al método que corresponde a cada test para que el mismo quede ligado a esa annotation. Por ejemplo:

```
@Test  
@Tag("smoke")  
public void testSearch1() {  
    //Search test  
}
```

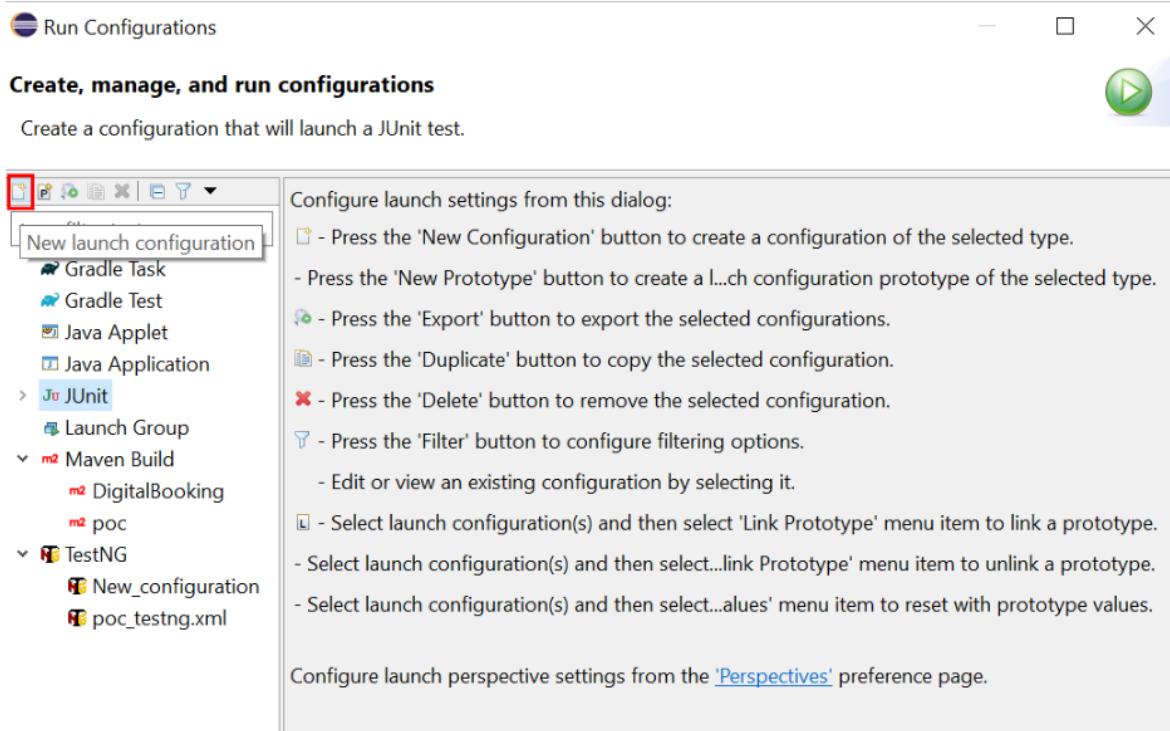
En el caso anterior, el caso de prueba “testSearch1” fue agrupado bajo la suite “smoke”. Luego de agregar ese tag a todos los casos de prueba que correspondan a la suite de smoke test, se puede ejecutar esa suite agregando el tag “smoke”.

Para la ejecución de los test desde Eclipse se deben seguir los siguientes pasos:

1. Ir al menú Run - Run Configurations...

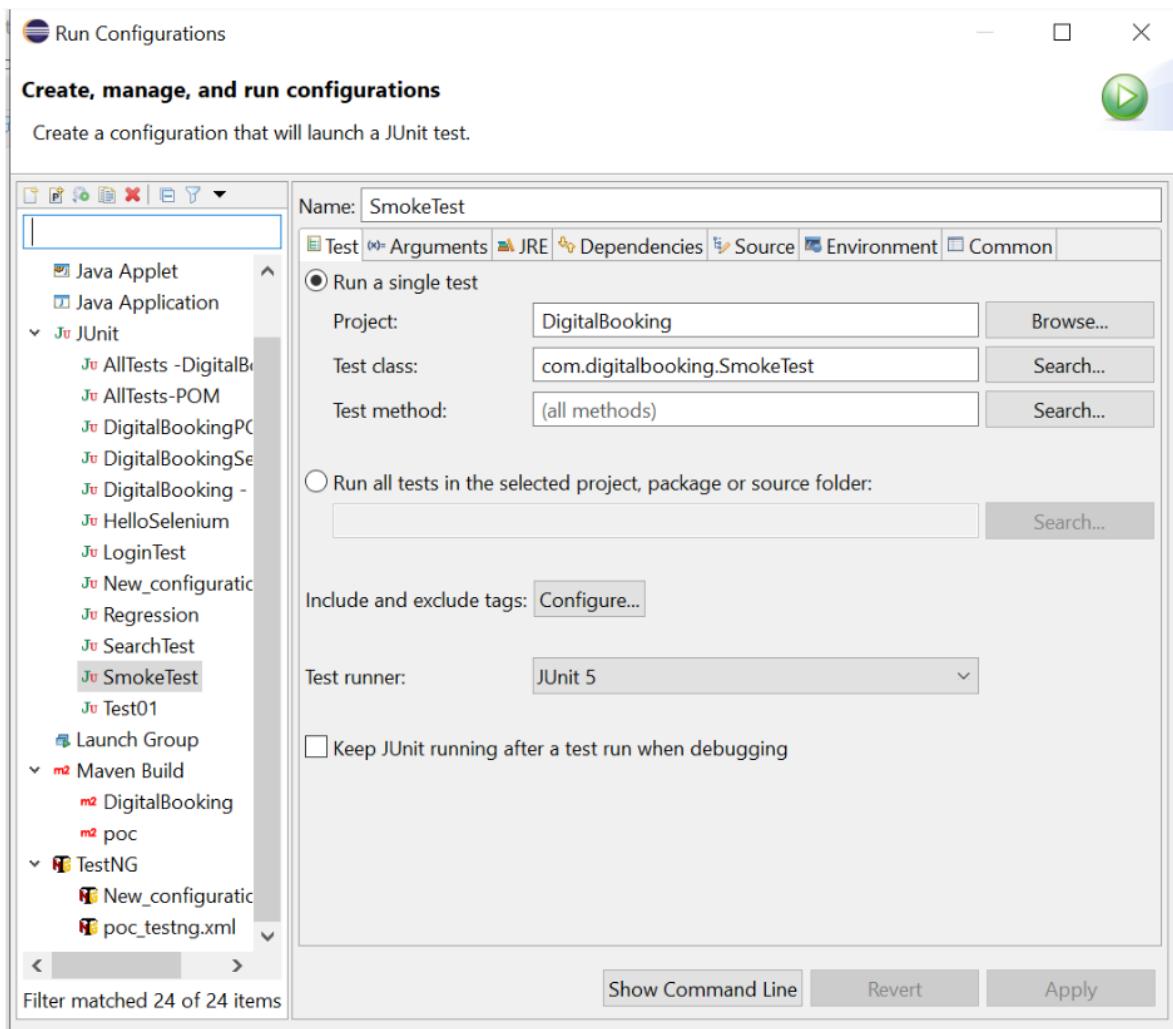


## Teniendo seleccionado JUnit, hacer clic en el ícono New launch configuration

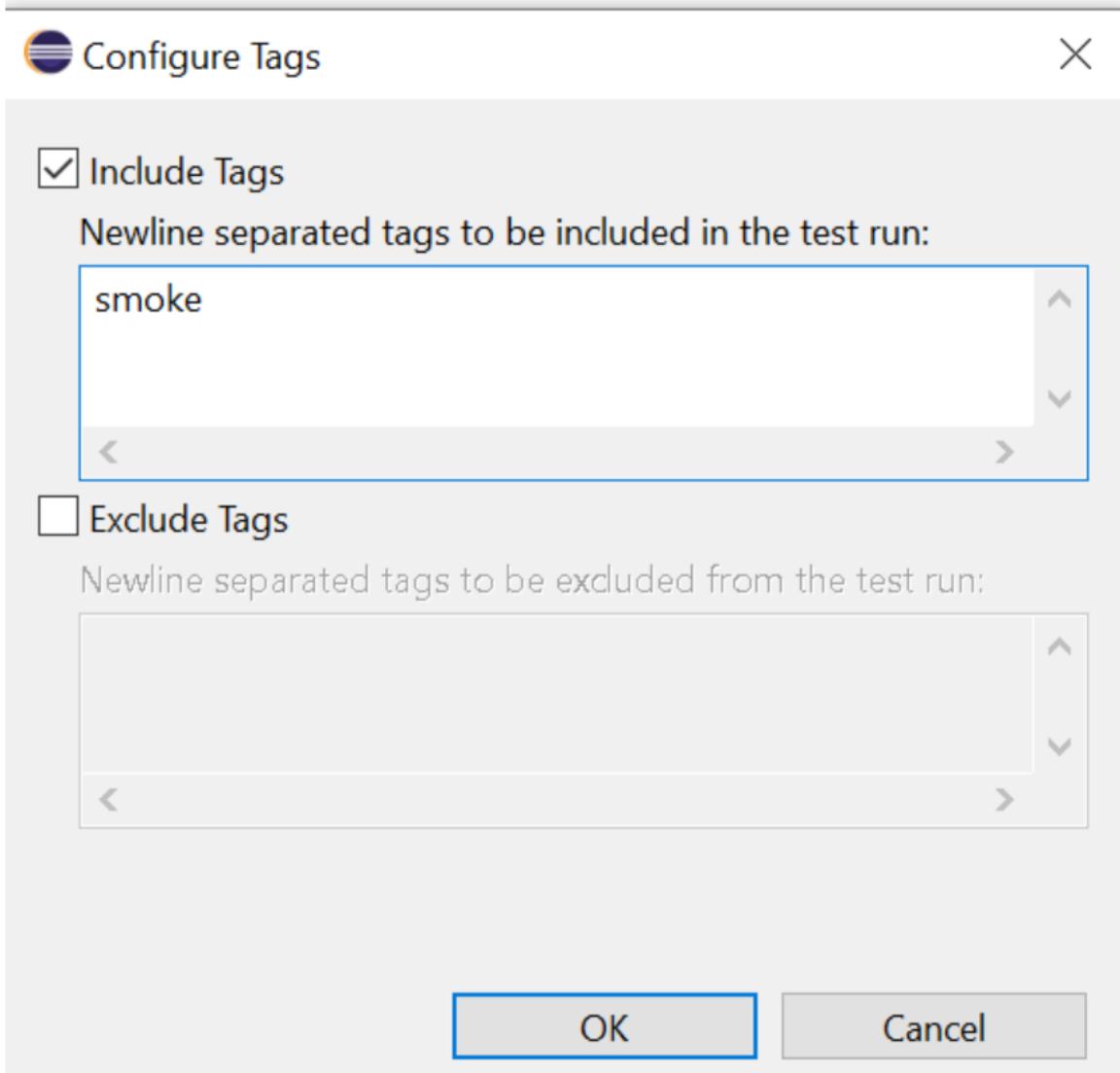


Ingresar los siguientes datos para esta nueva configuración:

- a. **Name** (Nombre)
- b. **Project** (Proyecto)
- c. **Test class** (Clase de prueba)
- d. Seleccionar en **Test runner** el ejecutor de Junit 5



Hacer clic en el botón **Configure** que corresponde a la etiqueta **Include and exclude tags**. En la ventana de configuración de Tags, seleccionar **Include Tag** e ingresar “smoke”. De esta manera se ejecutarán sólo los casos de prueba que tienen asignado ese tag.



**Tener en cuenta que también existe una opción para excluir test. Estas opciones pueden ser usadas en forma combinada para filtrar los casos de prueba que se desean ejecutar.**

Para comenzar la ejecución de los test, hacer clic en el botón **Run**.

## Generación de reportes para un Framework de automatización

La creación de un informe para una prueba permite que la prueba se complete al 100%. El informe no solo proporciona un resumen, sino que también:

- Ayuda a visualizar y resumir los resultados de la prueba
- Juega un papel vital en la automatización de pruebas

- Ayuda a los evaluadores y a las demás partes interesadas a comprender la estabilidad de las pruebas ejecutadas antes de que el producto entre en funcionamiento, brindando confianza para la puesta en marcha y posterior salida a producción

Un informe de prueba de automatización debe diseñarse de tal manera que **muestre los resultados de las pruebas al usuario final**. Un buen informe debe contener varias estadísticas de prueba, como el número total de casos de prueba de automatización, el número total de casos de prueba aprobados, el número total de casos de prueba fallidos, el porcentaje de casos aprobados y fallidos, clasificación de las pruebas según ciertos criterios, etc.

En los últimos años, varias herramientas de informes como TestNG Reports, JUnit Reports, Allure Reports, etc. están disponibles para generar el informe de automatización de pruebas. De esos, Extent Reports es una de las mejores herramientas de informes para Selenium que se usa ampliamente en varias organizaciones y ha ganado una inmensa popularidad debido a sus características únicas como la personalización de informes, la integración con diferentes frameworks de prueba, la visualización de datos, etc. En esta clase sobre reportes en Selenium, veremos el reporte de prueba más utilizado: **Extend Report**.

## Introducción a Extend Report

Extend Report es una **biblioteca de código abierto** que se utiliza para generar informes de prueba en las pruebas de automatización. Ha sido más utilizado para la generación de informes que los informes incorporados en varios frameworks de prueba debido a sus funciones y su posibilidad de personalización. Es una biblioteca de informes simple pero poderosa que se puede integrar con el framework de automatización para generar el informe de prueba de automatización

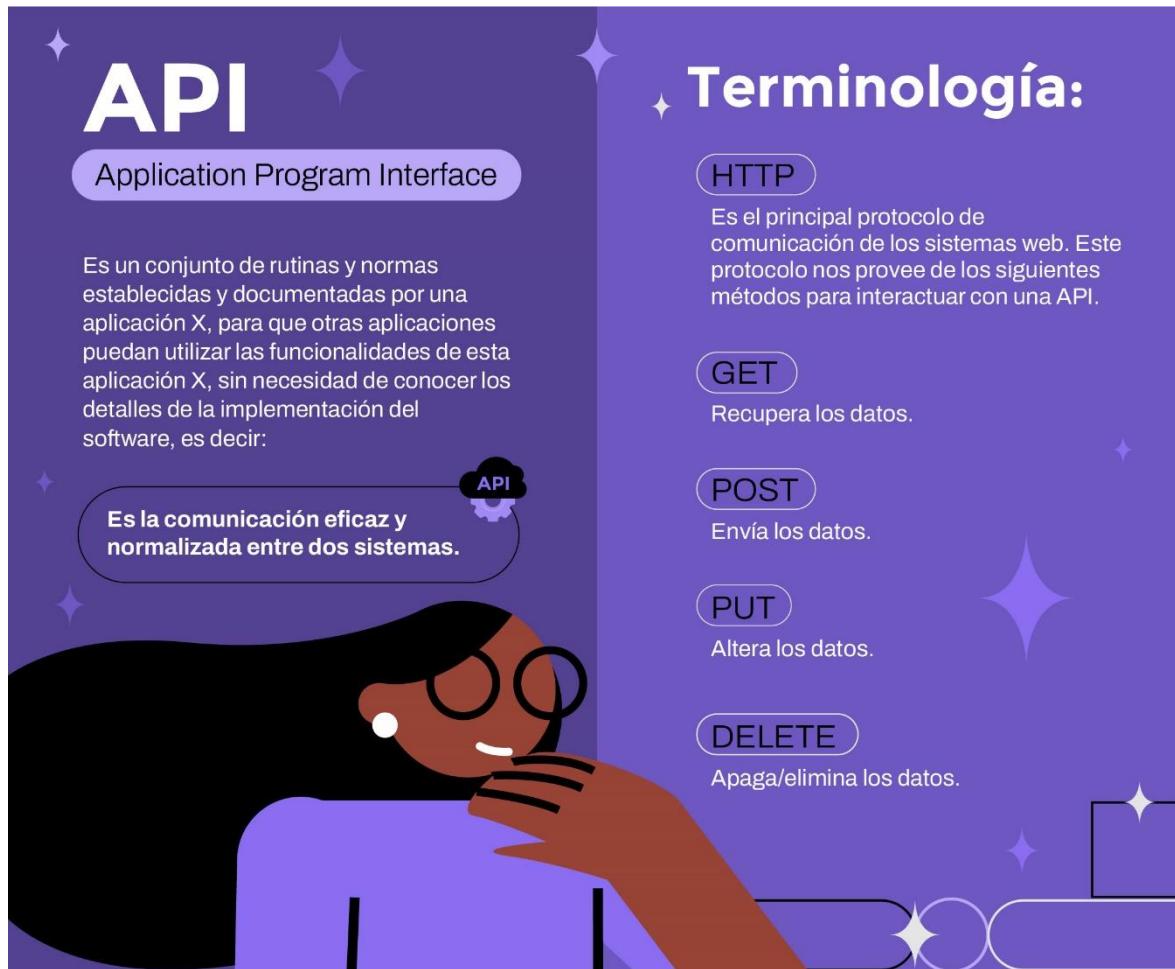
Los informes de extensión también se pueden integrar con otros marcos de automatización de pruebas populares como JUnit y TestNG. Proporciona una perspectiva detallada de los casos de prueba automatizados de forma gráfica. Es muy fácil integrar **Extend Report** con el framework de automatización que estamos construyendo...

## Ventajas de los informes de extensión

- Es una biblioteca de código abierto.
- Proporciona una representación pictórica de los resultados de la prueba.
- Se puede personalizar según se requiera.

- Permite a los usuarios adjuntar capturas de pantalla e iniciar sesión en el informe de prueba para obtener un resumen detallado de las pruebas.
- Se puede integrar fácilmente con otros marcos como JUnit, TestNG, NUnit, etc.
- Se puede configurar fácilmente con Jenkins, Bamboo, etc.

## API:



## ¿Qué es Rest Assured?

Rest-Assured es una **biblioteca Java de código abierto** que se utiliza para probar y validar las API de tipo REST. Los lenguajes dinámicos como Groovy y Ruby son útiles y sencillos para realizar pruebas de API que eran más difíciles en Java. Rest Assured se encarga de aportar simplicidad a la hora de realizar pruebas de APIs utilizando Ruby y Groovy en Java.

Rest Assured es un DSL de Java para simplificar las pruebas de los servicios basados en REST construidos sobre HTTP Builder. REST Assured admite cualquier método HTTP, pero tiene soporte explícito para POST, GET, PUT, DELETE, OPTIONS, PATCH y HEAD, e incluye la

especificación y validación, por ejemplo, de parámetros, cabeceras, cookies y cuerpo fácilmente.

Además, puede utilizarse para validar y verificar la respuesta a estas solicitudes.

## VENTAJAS de Rest-Assured



Utilizar Rest-Assured



Ventajas

- Es de código abierto, es decir, gratuito.
- La configuración inicial es sencilla antes de llegar a cualquier punto final.
- Fácil análisis y validación de la respuesta en JSON y XML.
- Sigue las palabras clave de BDD como given(), when(), then(), lo que hace que el código sea legible y permite una codificación limpia. Esta función está disponible a partir de la versión 2.0.
- Se pueden comprobar en tiempo real las cabeceras, las cookies, el tipo de contenido, etc.
- Puede integrarse fácilmente con otras librerías Java como TestNG, Junit como Test Framework y Extent Report y Allure Report para la elaboración de informes.



## Desventajas de Rest-Assured



Utilizar Rest-Assured



Ventajas Desventajas

- No admite la comprobación de las API SOAP de forma explícita.
- Requiere buenos conocimientos de programación en Java.
- No hay informes incorporados. Serenity BDD es una buena opción en este caso.



# ¿Cómo funciona Rest-Assured?

Entendamos cómo funciona Rest Assured:

- Crear una solicitud HTTP con todos los detalles
- Enviar la solicitud a través de la red
- Validar la respuesta recibida

## HTTP Request

Una HTTP request consiste en lo siguiente:



1. **URL** - La URL de solicitud es la dirección única utilizada para realizar una solicitud. La URL está compuesta por la URL base, el recurso (resource), la consulta (query) o los parámetros de la ruta (path parameters).
2. **HTTP Method/Verb** - Las cuatro operaciones básicas de creación, lectura, actualización y eliminación (CRUD) se realizan mediante los métodos POST, GET, PUT y DELETE en la interfaz REST.
3. **Headers** - Representa los metadatos asociados a las peticiones o respuestas HTTP. Es la información adicional que se pasa entre el cliente y el servidor junto con la solicitud o la respuesta. Las cabeceras se utilizan para diversos fines, como la autenticación, el almacenamiento en caché, la información del cuerpo del mensaje, la gestión de las cookies, etc. Las cabeceras serán pares clave-valor o pueden ser una clave con varios valores.
4. **Payload/Body** - Contiene la información que el usuario quiere enviar al servidor. La carga útil se utiliza únicamente con las solicitudes que modifican el recurso existente o crean otros nuevos.

5. **HTTP Response** - La respuesta HTTP consiste en el código de estado, las cabeceras (headers) y el cuerpo de la respuesta (response body).

6. **Códigos de status (status code)** - Los códigos de estado HTTP nos ayudan a comprender rápidamente el estado de la respuesta.

100-199: Informativo. Se ha recibido la solicitud y el proceso está en marcha.

200-299: Exitoso. La solicitud fue exitosa.

300-399: Redirección. La petición se está redirigiendo a otra URL.

400-499: Error del cliente. Se ha producido un error en el lado del cliente.

500-599: Error del servidor. Se ha producido un error del lado del servidor.

## ¿Cómo probar la API Rest?

Cuando se prueba un recurso REST, suele haber unas cuantas responsabilidades ortogonales en las que deben centrarse las pruebas:

- El status code HTTP
- Headers HTTP en la respuesta
- Payload (JSON, XML)

Cada prueba debe centrarse en una sola responsabilidad e incluir una sola afirmación. Centrarse en una separación clara siempre tiene ventajas, pero cuando se hace este tipo de pruebas de caja negra es aún más importante, ya que la tendencia general es escribir escenarios de prueba complejos desde el principio.

Ejemplo de prueba para comprobar el código de estado:

```
● ● ●

@Test
public void givenUserDoesNotExist_whenUserInfoIsRetrieved_then404IsReceived()
throws ClientProtocolException, IOException {

    // Given
    String name = RandomStringUtils.randomAlphabetic( 8 );
    HttpUriRequest request = new HttpGet( "https://api.github.com/users/" + name );

    // When
    HttpResponse httpResponse = HttpClientBuilder.create().build().execute( request );

    // Then
    assertThat(
        httpResponse.getStatusLine().getStatusCode(),
        equalTo(HttpStatus.SC_NOT_FOUND));
}
```

Esta es una prueba bastante simple. Comprueba que una ruta básica está funcionando, sin añadir demasiada complejidad al conjunto de pruebas. Si por la razón que sea falla, no es necesario examinar ninguna otra prueba para esa URL hasta que se solucione.

Esto es sólo una parte de lo que debería ser un conjunto de pruebas de integración completo. Las pruebas se centran en garantizar la corrección básica de la API REST, sin entrar en escenarios más complejos, por ejemplo, no se contemplan los siguientes escenarios: descubrimiento de la API, consumo de diferentes representaciones para un mismo recurso, etc.

La implementación de este ejemplo y el fragmento de código se pueden encontrar en Github - este es un proyecto basado en Maven, por lo que debería ser fácil de importar y ejecutar tal cual.

## Conclusión

Dado que las pruebas de API son una parte fundamental de cualquier ciclo de vida de desarrollo, REST Assured Framework es una de las herramientas de prueba de servicios web de Java más utilizadas. Las características avanzadas, junto con la simplicidad en la implementación, hacen que sea una necesidad para cualquier tester para garantizar la calidad del producto final.

## Testing II

# Vamos a instalar Rest Assured

### Pre-requisito:

Para iniciar el proyecto, deben tenerlo instalado y configurado en su ordenador:

- JDK 18
- Maven 3.8.x
- Eclipse IDE

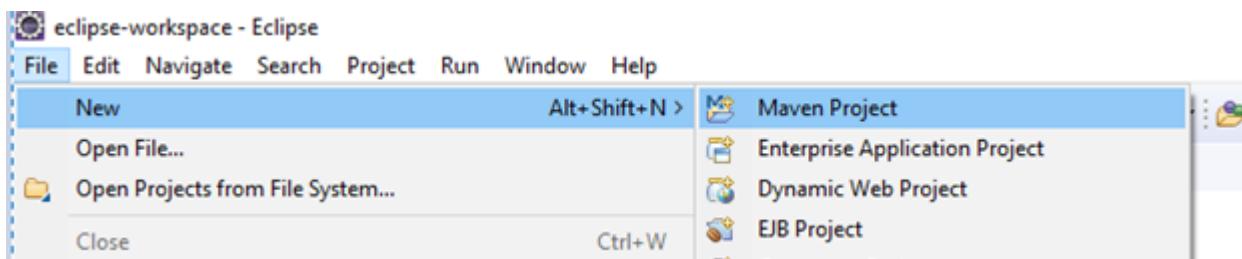
### Pasos:

Comencemos con la creación del proyecto.

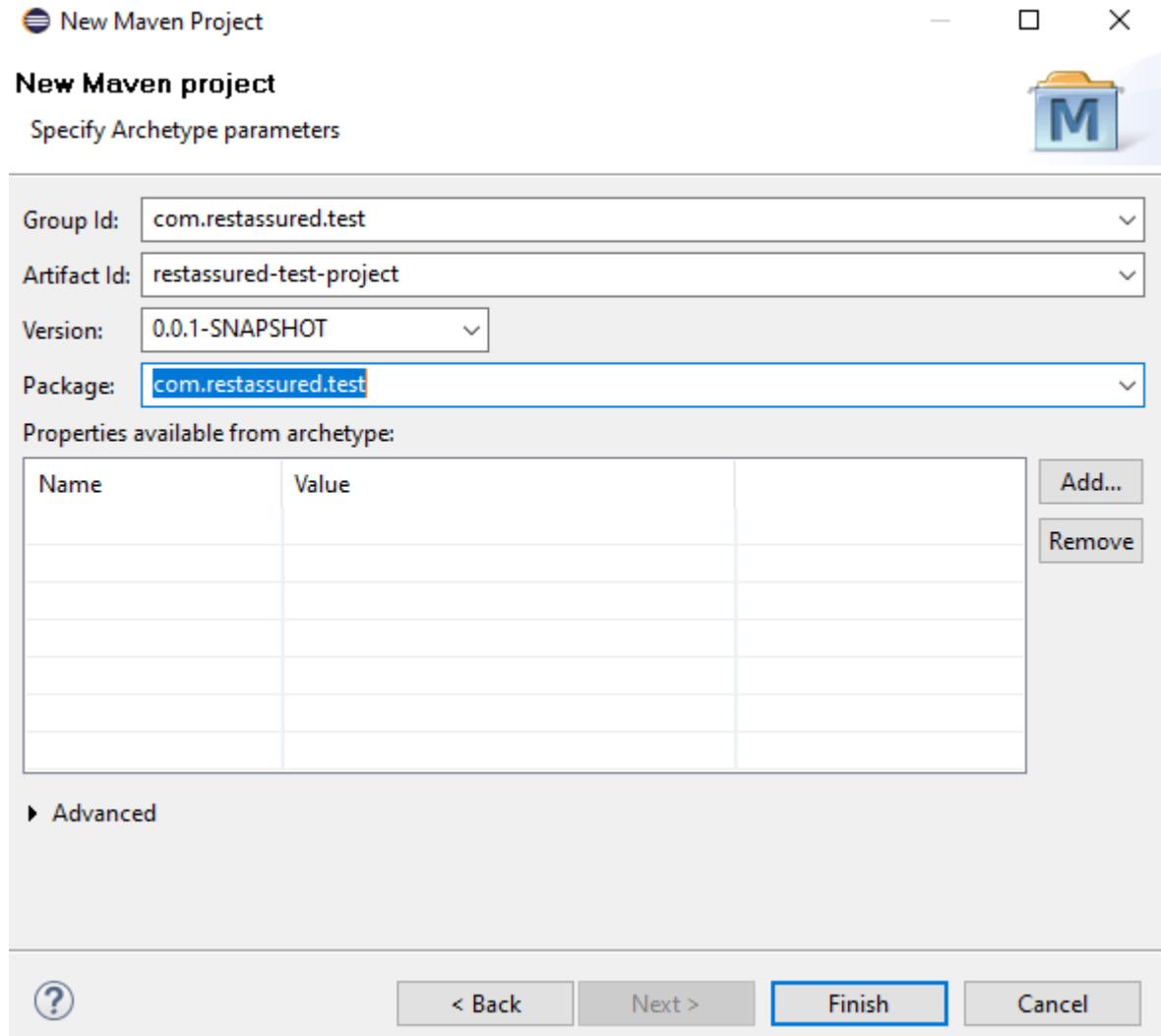
#### - Creación de un nuevo proyecto en Eclipse

Empecemos por la creación del proyecto. Abrimos el IDE de Eclipse y creamos un nuevo proyecto.

1. Clic en File → New → Maven Project



2. Establezcamos el espacio de trabajo y hagamos clic en Siguiente
3. Seleccionemos el arquetipo como maven-archetype-quickstart y hagamos clic en Siguiente
4. Proporcionemos los detalles del ID del dispositivo, el ID del grupo, el nombre y la descripción. A continuación, hagamos clic en Finish



## Configurar Junit:

1. En el archivo pom.xml, añadamos el REST Assured para maven  
T

## Última dependencia de JUnit Maven a partir de ahora

```
<dependencies>
    [...]
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>5.4.0</version>
        <scope>test</scope>
    </dependency>
    [...]
</dependencies>
```

2. Creemos el archivo XML del conjunto de pruebas para llamar a la clase de prueba y pongamos el siguiente código, por ejemplo, testng.xml en la carpeta src\test\resources
3. Establezcamos la configuración del plugin Surefire para ejecutar el archivo xml

```
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-surefire-plugin</artifactId><version>2.20</version>
<configuration>
<suiteXmlFiles><suiteXmlFile>src\test\resources\testng.xml</suiteXmlFile>
</suiteXmlFiles>
</configuration>
</plugin>
```

### Configurar REST Assured

En el archivo pom.xml, adicionemos a REST Assured para maven

URL — <https://mvnrepository.com/artifact/io.rest-assured/rest-assured>

### Última dependencia de Maven Rest Assured a partir de ahora.

```
<!-- https://mvnrepository.com/artifact/io.rest-assured/rest-assured -->
<dependency>
    <groupId>io.rest-assured</groupId>
    <artifactId>rest-assured</artifactId>
    <version>4.1.2</version>
    <scope>test</scope>
</dependency>
```

## Escribir el método de prueba

1. Crear una clase llamada "SampleTest.java".
2. Añadir el siguiente método a la clase

```
@Test
public void sampleLogin() {
    given().contentType("application/x-www-form-urlencoded; charset")
        .formParam("grant_type", "password")
        .formParam("username", "Test User")
        .formParam("password", "Test123&&")
        .when().post("http://demo6116845.mockable.io/login")
        .then().statusCode(200);
}
```

Ejecutar el proyecto de prueba utilizando el comando **mvn clean surefire-report:report**

¡Listo! Han creado su primera prueba REST Assured.

# Rest Assured

- Es una biblioteca Java o API para probar servicios web RESTful.
- Se utiliza para probar servicios web basados en XML y JSON.
- Admite diferentes métodos HTTP: GET, POST, PUT, DELETE, OPTIONS, PATCH, HEAD.
- Se puede integrar con marcos de prueba como JUnit, TestNG.



# GET

1. Lo primero que debemos hacer es realizar la petición GET al endpoint correspondiente y lo almacenamos en una variable.
2. Como vemos en este apartado podemos mostrar diferente información obtenida desde el endpoint consultado.
3. El siguiente paso es realizar las validaciones necesarias para verificar que la información obtenida sea la deseada, validar los error code HTTP, etc. Podemos utilizar diferentes [Assert\(\)](#) para validar la información.

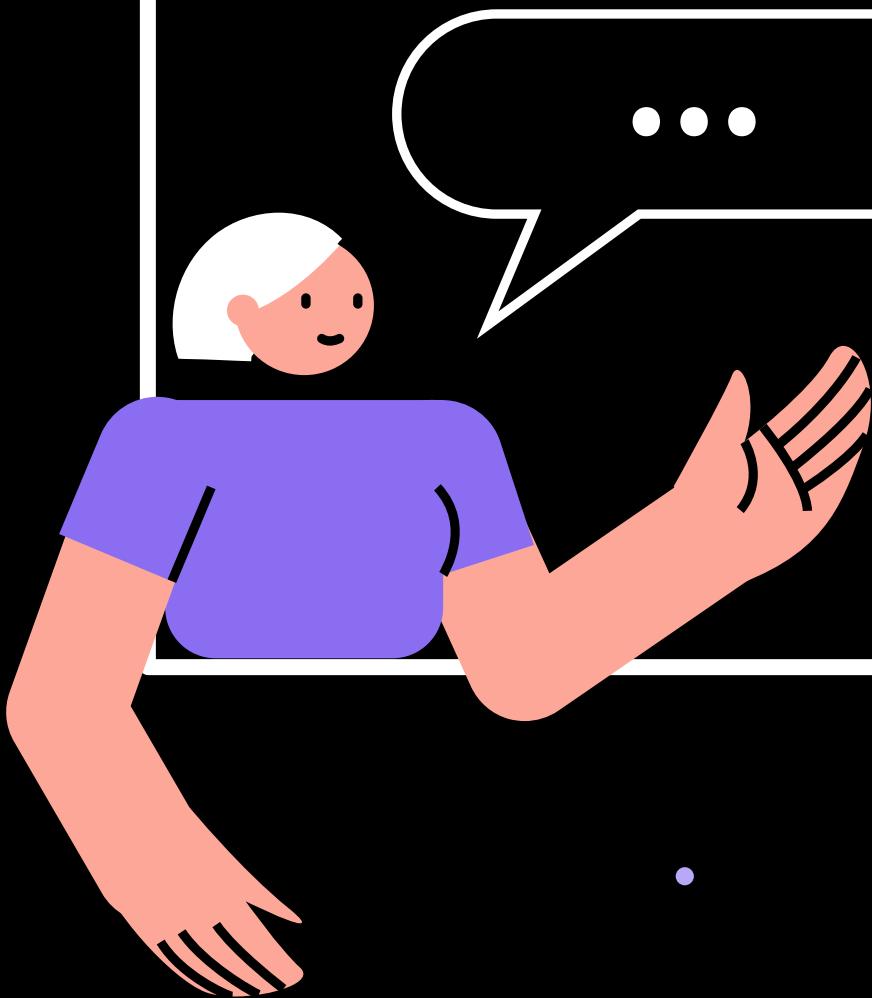
```
public class Test01_GET {  
  
    @Test  
  
    void test01() {  
  
        Response response =  
①         RestAssured.get("https://reqres.in/api/users?page=2");  
  
        System.out.println(response.getBody().asString());  
        System.out.println(response.asString());  
②        System.out.println(response.getStatusCode());  
        System.out.println(response.getHeader("content-type"));  
        System.out.println(response.getTime());  
  
        // Validar status code  
        int statusCode = response.getStatusCode();  
③        Assert.assertEquals(statusCode, 200);  
  
        // Validar contenido body  
        String body = response.getBody().asString();  
        Assert.assertEquals(body.data.id[0].email,  
                           "michael.lawson@reqres.in");  
    }  
}
```

# Rest Assured+Gherkin

Rest Assured nos permite utilizar Gherkin. Gherkin define 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.

Sólo nos hace falta conocer 5 palabras para empezar a utilizar BDD.

- **FEATURE:** Nombre de la funcionalidad que vamos a probar
- **SCENARIO:** Describe cada escenario que vamos a probar.
- **GIVEN:** Contexto para el escenario en que se va a ejecutar el test.
- **WHEN:** Conjunto de acciones que lanzan el test.
- **THEN:** Especifica el resultado esperado en el test.



# GET

1. Podemos utilizar la nomenclatura de Gherkin para escribir nuestros tests. De manera que nuestro código sea más legible por otras personas.

```
public class Test01_GET {  
    @Test  
    void test01() {  
        given().  
        ① get("https://reqres.in/api/users?page=2").  
        then().  
        statusCode(200).body("data.id[0]", equalTo(7));  
    }  
}
```

# POST

1. Lo primero que debemos hacer es crear un objeto del tipo `JSONObject()`, el cual enviará toda la información que viajará por el body de nuestra request.
2. Como vemos, estamos utilizando Gherkin, de manera tal que en nuestro `given()` vamos a configurar todos los parámetros necesarios para enviar nuestra petición.
3. Hacemos una petición POST a nuestra API.
4. Finalmente realizamos todas las validaciones necesarias para verificar que la información obtenida sea la deseada, validar los error code HTTP, etc

```
public class Test01_POST {  
  
    @Test  
  
    void test01() {  
  
        Jt request = new JSONObject();  
        ① request.put("name"SONOjje, "Ajeet");  
        request.put("Job", "Teacher");  
  
        given().  
            header("Content-type", "application/json").  
            contentType(MediaType.APPLICATION_JSON).  
            body(request.toJSONString());  
  
        ② when().  
            post("https://reqres.in/api/users").  
            then().  
            ③ statusCode(201).log().all();  
  
        ④ }  
    }  
}
```

# PUT

1. Lo primero que debemos hacer es crear un objeto del tipo `JSONObject()`, el cual enviará toda la información que viajará por el body de nuestra request.
2. Configuramos todos los parámetros necesarios para enviar nuestra petición.
3. Hacemos una petición PUT a nuestra API.
4. Finalmente realizamos todas las validaciones necesarias para verificar que la información obtenida sea la deseada, validar los error code HTTP, etc.

```
public class Test01_PUT {  
    @Test  
    void test01() {  
        JSONObject request = new JSONObject();  
        ① request.put("name", "Amann");  
        request.put("Job", "Teacher");  
        ② given().  
            header("Content-type", "application/json").  
            contentType(MediaType.APPLICATION_JSON).  
            body(request.toJSONString());  
        when().  
        ③ put("https://reqres.in/api/users/2").  
        then().  
        ④ statusCode(201).log().all();  
    }  
}
```

# DELETE

1. Realizamos la petición DELETE al endpoint correspondiente.
2. Finalmente realizamos todas las validaciones necesarias para verificar que la información obtenida sea la deseada, validar los error code HTTP, etc.

```
public class Test01_DELETE {  
    @Test  
    void test01() {  
        given().  
        ① delete("https://reqres.in/api/users/2").  
        then().  
        ② statusCode(204).log().all();  
    }  
}
```

La adopción de la arquitectura de microservicios ha aportado flexibilidad, escalabilidad, reducción de costos y tiempo de desarrollo de sistemas. Este tipo de arquitectura presupone la división de la aplicación en partes y, en consecuencia, la comunicación entre ellas. Así, utilizamos las API para crear esta comunicación y establecer el consumo de información de forma rápida y segura. Comúnmente utilizado en aplicaciones web que presentan un alto volumen de intercambios de datos procesados de forma asíncrona entre varias aplicaciones.

Las API pueden clasificarse como API REST o API RESTful en función de su grado de adhesión a los requisitos, reglas y principios de la arquitectura REST. Al adoptar este estilo arquitectónico, la manipulación de los recursos se realiza a través de los métodos del protocolo HTTP.

## Testing en APIs

Teniendo en cuenta el contexto, es crucial probar la funcionalidad, la fiabilidad, el rendimiento y la seguridad de las API. Para ello, hay una serie de herramientas que apoyan estas pruebas, ya sean manuales o automatizadas. Postman, por ejemplo, es una de las herramientas que vimos en Testing I.

Se pueden realizar algunas pruebas para:

- Validar los datos de retorno.
- Validar los headers de respuesta.
- Validar que la respuesta es conforme.
- Validar el comportamiento después de cambiar el tipo de contenido.
- Validar si la estructura JSON o XML es correcta.
- Validar si el estado es acorde a los códigos de error.
- Validar el comportamiento de la solicitud con información incompleta.

## Rest Assured

Rest Assured es una de las herramientas de código abierto basadas en Java que se utilizan para simplificar las pruebas de los servicios REST. Permite crear llamadas HTTP y utilizar métodos y utilidades para validar la respuesta recibida del servidor, como el código de estado y otra información. Además, se puede utilizar junto con JUnit.

En el siguiente código tenemos la creación del script usando Rest Assured.

```
import io.restassured.RestAssured;
import io.restassured.http.Method;
import io.restassured.response.Response;
import io.restassured.specification.RequestSpecification;

public class RestAssuredAPITest {

    @Test
    public void GetBooksDetails() {
        // Specify the base URL to the RESTful web service
        RestAssured.baseURI = "https://demoqa.com/BookStore/v1/Books";
        // Get the RequestSpecification of the request to be sent to the server.
        RequestSpecification httpRequest = RestAssured.given();
        // specify the method type (GET) and the parameters if any.
        //In this case the request does not take any parameters
        Response response = httpRequest.request(Method.GET, "");
        // Print the status and message body of the response received from the server
        System.out.println("Status received => " + response.getStatusLine());
        System.out.println("Response=>" + response.prettyPrint());
    }
}
```

# Métodos HTTP

Los métodos identifican los diferentes tipos de operaciones que se pueden realizar para manipular los datos. En general, los métodos principales representan operaciones CRUD (crear, leer, actualizar, borrar).

Los códigos de estado HTTP se utilizan como respuesta a la solicitud, para indicar el estado/resultado de la misma. Estos códigos se agrupan en diferentes grupos: Informal, Éxito, Redirección, Error del Cliente y Error del Servidor.

Compruebe a continuación los principales métodos, sus significados y los códigos de respuesta:

Verbo HTTP	Descripción	Respuesta
POST	Se utiliza para crear un nuevo registro en la base de datos	Status code 201 - registro creado
GET	Se utiliza para leer registros de la base de datos	Status code 200 - devuelve los registros en el formato solicitado
PUT	Se utiliza para actualizar un registro en la base de datos	Status code 200 - registro actualizado
PATCH	Se utiliza para actualizar parte de un registro en la base de datos	Status code 200 - registro actualizado
DELETE	Se utiliza para eliminar un registro en la base de datos	Status code 204 - registro eliminado

¡Comprueben [aquí](#) los demás códigos de forma creativa!

El uso de API's en el desarrollo de sistemas es cada vez más común, principalmente debido a la adopción de microservicios. Por lo tanto, es esencial que probemos cómo se comportan estas API, si cumplen los requisitos, si son seguras, etc. Para ello, hay una serie de herramientas que soportan estas pruebas, que pueden ser manuales o no. Una de estas herramientas es Rest Assured, de código abierto y basada en Java, que permite simplificar la creación de pruebas automatizadas para la validación de la API.

En general, funciona creando llamadas HTTP y validando su retorno, es decir, toda la información recibida como respuesta del servidor. Una de estas informaciones es el código de estado: los más comunes son los relacionados con las operaciones CRUD, identificados por los verbos HTTP (POST, GET, PUT, PATCH y DELETE). Para empezar a usarlo, hay que declarar la dependencia en el archivo POM.xml (proyectos que usan Maven) y también importar estáticamente en la clase de prueba.

Cabe mencionar que cuando se trata de probar API's es fundamental que antes de escribir los casos de prueba, se consulte el contrato de la API a través de su documentación, para entender los endpoints disponibles y sus características generales.

## Integración Continua

El software es un viaje de interacción en el que la innovación viene acompañada de una gran cantidad de pruebas y errores. A medida que avanzamos hacia la producción, nuestra confianza crece.

La integración continua (CI) es una parte esencial de esta iteración, ya que permite realizar construcciones sistémicas y publicar artefactos de despliegue en repositorios de artefactos.

El ciclo de vida de la entrega de software (SDLC) es un ejercicio de construcción de confianza, y a medida que aumenta la propiedad, los equipos de desarrollo de software quieren producir características y artefactos de alta calidad, y los pasos de ingeniería de calidad siguen avanzando para aumentar la confianza.

Un punto natural para comenzar a realizar ciertos pasos de construcción de confianza y asegurar la calidad y consistencia de los cambios es el pipeline de Integración Continua (CI).

## Integración continua (Continuous Integration)

La integración continua (CI) es un **proceso automatizado para integrar continuamente los cambios en el desarrollo de software**. Los procesos de CI automatizan la construcción, las pruebas y la validación del código fuente.

Al trabajar con las capacidades de CI, los desarrolladores pueden acelerar sus ciclos de liberación de código, disminuyendo la probabilidad de enfrentarse a largos ciclos de desarrollo y los desafíos que vienen con ellos, como los conflictos de fusión y el control de versiones.

La Integración Continua **requiere un sistema de control de versiones que pueda rastrear los cambios y las versiones del código del software**. Git es un popular sistema de control de versiones y utilizando un flujo de trabajo git, se puede iniciar el proceso de CI.

Los desarrolladores que trabajan en una base de código utilizan sus sistemas de control de versiones para enviar sus cambios a un repositorio de código y fusionar estos cambios en la rama de código principal después de que haya sido revisada.

El objetivo de CI es **producir un artefacto empaquetado que esté listo para ser desplegado en un servidor o máquina**. Un ejemplo de artefacto puede ser una imagen de contenedor, un archivo WAR/JAR o cualquier otro código ejecutable empaquetado. CI automatiza el proceso desde la confirmación del código hasta el artefacto empaquetado.

Por lo general, un desarrollador envía algún código a un sistema de control de versiones como Git, lo que desencadena el proceso de CI. Esta base de código se suele escanear o analizar con una herramienta de análisis estático de código para determinar la calidad del mismo. **Si el código fuente pasa todas las comprobaciones**, incluidas las pruebas unitarias, **el proceso de CI intentará empaquetar o compilar el código**. La implementación de un proceso de integración continua puede ser el primer paso para producir código de alta calidad en sus equipos. Hay muchas **herramientas de integración continua** disponibles, entre las más populares está **Jenkins**.

En el campo de la ingeniería de software, la iteración es la clave de la innovación. El ingeniero de software construirá y creará varias veces al día en su máquina local, en algún momento sus ideas necesitan llegar a equipos más amplios, y aquí es donde entra en juego la integración continua (CI).

Sin embargo, en una compilación se puede incluir algo más que el código fuente compilado. El producto final de la integración continua es una versión candidata. Una versión candidata es la forma final de un artefacto para ser desplegado. Es posible que se tomen medidas de calidad para producir el artefacto, como encontrar bugs e identificar su solución. El empaquetado, la distribución y la configuración van en una versión candidata.

## ¿Por qué utilizar la integración continua?

Tener un artefacto o artefactos listos para el despliegue para seguir comprobando (por ejemplo, desde el desarrollo hasta el entorno de garantía de calidad) es prudente en las organizaciones de ingeniería de software actuales. El núcleo del trabajo de un ingeniero de software es de naturaleza iterativa. Es posible que se creen varios artefactos antes de hacer una versión candidata viable. La capacidad de construir bajo demanda e iniciar el camino de la integración y la calidad comienza con una construcción que puede ocurrir varias veces al día. Según Paul Duvall, coautor de Continuous Integration, en pocas palabras, la IC mejorará la calidad y reducirá el riesgo.

## Ventajas de la integración continua

Tener un enfoque de integración continua libera a los equipos de la carga de las construcciones manuales y también hace que las construcciones sean más repetibles, consistentes y disponibles. Tener el principal producto de trabajo de un equipo de ingeniería de software (la unidad desplegable) listo para ser desplegado de forma regular es beneficioso para todo el ciclo de vida de la entrega de software y permite una colaboración consistente entre los ingenieros, evitando cuellos de botella comunes.

### Repetibilidad

La subcontratación de la construcción en lugar de ser bloqueada localmente por un desarrollador pone más ojos en los pasos de construcción. La configuración de la integración continua empieza a tener un enfoque menos individual. **Tener una construcción ejecutada por un sistema es una marcha hacia tener consistencia.**

La capacidad de construir de forma consistente es uno de los principales pilares de la integración continua. Una vez que se tienen construcciones repetibles, la eficiencia y la consistencia comienzan a mostrarse a medida que las prácticas de Integración Continua se vuelven más maduras en el equipo. Con la consistencia, también existe la posibilidad de tener más construcciones disponibles.

### Disponibilidad

La capacidad de escalar para satisfacer las demandas de las construcciones concurrentes requeridas en un equipo y la capacidad de recrear una construcción es lo que denominamos disponibilidad de construcción. Las construcciones modernas en contenedores requieren más potencia que la simple construcción del binario de la aplicación. **Los sistemas de compilación distribuidos permiten que estas compilaciones estén más disponibles.** Dado que las construcciones son repetibles y consistentes, un principio básico del desarrollo de software moderno es ser repetible en cualquier paso del proceso. Se puede acceder a las construcciones antiguas y a las versiones anteriores simplemente llamando a una versión del pasado. Con el énfasis puesto en tener una construcción disponible en cualquier momento, pueden surgir desafíos en el apoyo a una amplia gama de tecnología.

## Prueba de integración continua

Las pruebas de integración continua son **pruebas enfocadas y ejecutadas durante el proceso de CI** y orquestadas por las herramientas de CI que son responsables de construir, empaquetar y publicar artefactos. Hay un cierto nivel de calidad que se espera cuando un artefacto o más artefactos se están construyendo y viajan por el camino hacia una versión candidata.

Las pruebas en el proceso de IC **permiten una rápida retroalimentación y, por diseño, detienen el progreso del artefacto si no se cumple la calidad mínima.** Normalmente, las pruebas de CI se centran en el artefacto antes del despliegue en el primer entorno de integración.

La repetibilidad es un concepto crucial en la creación y construcción de software. La posibilidad de crear o recrear en cualquier momento aumenta la fiabilidad. Como ya vimos, uno de los grandes beneficios de la integración continua es la repetibilidad; la externalización de las compilaciones frente a su bloqueo local en la máquina de un desarrollador.

Otro punto importante es la capacidad de ejecutar construcciones o pruebas en paralelo con otros miembros del equipo. La cobertura de la prueba que se ejecuta localmente es un excelente candidato para ser dirigido a la tubería de integración continua. Al igual que DevOps, que reúne a los equipos de desarrollo y de operaciones, las pruebas en el canal de CI pueden ayudar a los equipos de control de calidad a pasar al equipo de desarrollo de forma sistemática.

## La importancia de las pruebas de CI

El desarrollo local puede incurrir en docenas de construcciones y ciclos antes de un commit, aunque la integración de las características (nuevas funcionalidades) recién escritas o revisadas en la aplicación/servicio comienza con la construcción. Pasar las pruebas de integración continua no sólo significa que el artefacto tiene la capacidad de ser construido de manera consistente, sino que también se ha logrado un nivel de calidad en torno al artefacto antes de la liberación.

Una construcción fallida es mucho menos grave que un despliegue fallido. Que se encuentren errores y correcciones en las pruebas de CI es perfectamente normal. Estas pruebas se orquestan en una canalización y se dividen en dos categorías: pruebas dentro y fuera del proceso.

## Tipos de pruebas de CI

Desde el inicio del proceso de CI (es decir, que se desencadena por un check-in de código) hasta un artefacto publicado, hay varios tipos de pruebas de CI que se realizan durante la compilación automatizada.

- **Pruebas de calidad del código**

Las herramientas de calidad del código, como SonarQube y Checkmarx , se centran en el análisis estático del código y son prudentes durante los cambios de código. El código por sí mismo no se ejecuta y tiene en cuenta toda la infraestructura y las variables de entorno que alimentan el software en ejecución, aunque el análisis del código puede inferir la calidad. La comprobación de los bloques muertos (nunca llamados o alcanzados en el código), la calidad de la sintaxis/patrones y los posibles problemas de seguridad forman parte de las pruebas de calidad del código. Mientras que la calidad del código no se centra en la funcionalidad real del código, aquí es donde entran en juego las pruebas unitarias.

- **Pruebas unitarias**

Las pruebas unitarias se centran en los bloques/métodos de código que han cambiado. Si está construyendo un nuevo proyecto desde cero, cubrirían la funcionalidad de la aplicación. Las pruebas unitarias suelen ser pruebas de proceso en las que se crean objetos simulados y se comprueban las afirmaciones. JUnit en el mundo JAVA y Mochano NPM. Las pruebas unitarias están diseñadas para ser granulares y ejecutarse como un conjunto. Por ejemplo, si está escribiendo una aplicación de calculadora y añadiendo una nueva función para dividir enteros, una prueba de unidad podría ser cómo se maneja la división por cero o el resultado esperado de ejecutar esa división. Si el caso de uso requiere llamar a métodos/partes externas, la cobertura de la prueba unitaria no será suficiente y la prueba de integración tomará el relevo.

- **Prueba de integración**

Las pruebas de integración pueden ser de gran alcance, dependiendo de los límites definidos en la prueba. En el caso de las pruebas de integración en el contexto de la integración continua, se centrará en probar módulos cruzados de la aplicación. Volviendo al ejemplo de la calculadora, una prueba de integración sería dividir y multiplicar al mismo tiempo. Las herramientas modernas de pruebas unitarias y de integración se solapan mucho. JUnit puede ser utilizado para pruebas unitarias y de integración porque JUnit puede perseguir/seguir/invocar llamadas a métodos que pueden ser encadenados.

- **Prueba de seguridad/licencia**

El dicho de que el software envejece más como la leche que como el vino es cierto: especialmente en el software moderno con dependencias de código abierto de terceros, el equipo de ingeniería rara vez es el autor del 100% de los bits que van a la distribución final. Con la "niebla del desarrollo", no sería razonable que un ingeniero conociera todas las dependencias transitivas (dependencias que llaman a otras dependencias). El objetivo de las pruebas de seguridad/licencias es averiguar la exposición y el riesgo de utilizar determinados paquetes. Las herramientas de análisis de seguridad/licencias como Blackduck , Snyk y StackHawk tienen diferentes métodos de introspección. Algunas herramientas requieren que se ejecute el artefacto terminado, por ejemplo una imagen docker.

- **Pipeline**

Un pipeline es una serie de pasos orquestados con la capacidad de llevar el código fuente a la producción. Los pasos incluyen la creación, el empaquetado, las pruebas, la validación, la verificación de la infraestructura y la implantación en todos los entornos necesarios. Dependiendo de las estructuras organizativas y de los equipos, pueden ser necesarias varias vías para lograr este objetivo. Una canalización puede ser activada por algún tipo de evento, como una solicitud de extracción de un repositorio de código fuente (es decir, un cambio de código), la presencia de un

nuevo artefacto en un repositorio de artefactos, o algún tipo de programación regular para que coincida con una cadencia de lanzamiento.



La continuidad es un rasgo distintivo de un pipeline. Esto incluye la integración continua, la entrega continua/despliegue continuo (CI/CD), la retroalimentación continua y las operaciones continuas. En lugar de pruebas puntuales o despliegues programados, cada función se produce continuamente.

## Principales pilares de la integración continua



### Construir

Ejecución de herramientas de compilación dependientes del lenguaje, como en el ecosistema JAVA, como Maven/Gradle, o en el ecosistema NodeJS, como NPM. Su compilación combina un código escrito a medida con dependencias de terceros.

### Probar

Las pruebas centradas en las unidades y en los artefactos son prudentes para ser ejecutadas durante la Integración Continua. Como el artefacto aún no ha sido desplegado, las pruebas unitarias son óptimas para ejecutar en el proceso de integración continua. Si es necesario probar el despliegue de la infraestructura o de la aplicación, por ejemplo, en una prueba de sistema o de integración, esto es más adecuado para la entrega continua. A medida que se construye o se ha construido el artefacto/unidad desplegable, las comprobaciones centradas en el artefacto, como las comprobaciones de código abierto y de dependencias, son excelentes formas de encontrar vulnerabilidades y posibles riesgos de licencia.

## Publicar

Una vez creado el artefacto/unidad, la publicación en un repositorio/registro de artefactos es una práctica estándar del sector. En el ejemplo de JAVA, se puede publicar el artefacto en un repositorio Maven/artifact y luego la imagen Docker en un registro Docker.

# Herramientas de integración continua

A medida que las construcciones se hicieron más frecuentes, un cuello de botella común de DevOps fue la infraestructura de construcción, especialmente cuando se trata de la contenerización. Las soluciones internas y las primeras plataformas de integración continua no se construyeron para la escala requerida en el mundo nativo de la nube de hoy.

Con el aumento de los microservicios, hay un orden de magnitud más de construcciones, y la potencia adicional necesaria para empaquetar sus construcciones se suma a la sobrecarga. Ciertamente hay muchas opciones de herramientas de integración continua, pero una de las más utilizadas es Jenkins, que se considera el pivote para hacer de la integración continua la corriente principal.

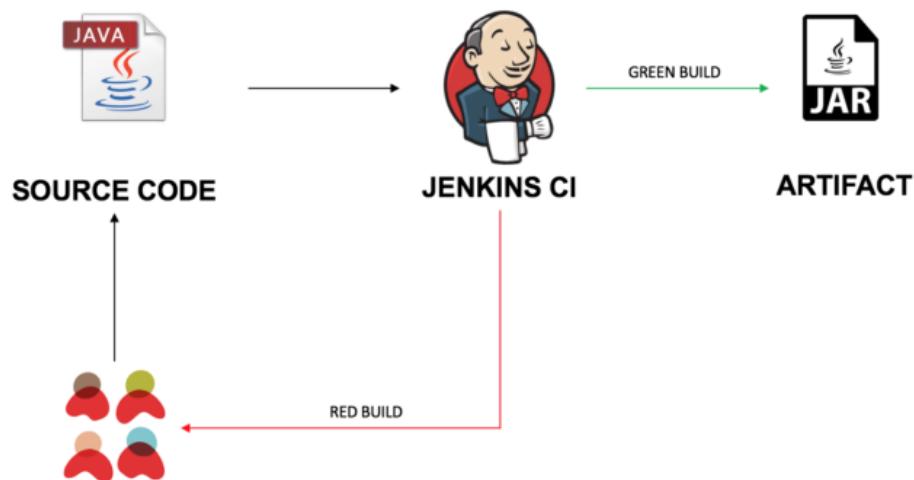
## Jenkins

Jenkins se ha convertido en una familia de productos e interpretaciones desde su cambio de nombre en 2011. Viniendo del proyecto Hudson, Jenkins se centró en las construcciones JAVA para empezar.

Con el tiempo, se creó Jenkins X para solucionar las limitaciones de escalado de la plataforma. Dependiendo de la versión, puede ser necesario el soporte de más de un lenguaje (por ejemplo: DSL ((domain-specific-language)) Pipeline o scripts escritos en Groovy). Jenkins es una solución de código abierto que cuenta con el apoyo de proveedores, principalmente Cloudbees.

Jenkins **es un servidor de integración continua de código abierto escrito en Java** para orquestar una cadena de acciones para lograr el proceso de integración continua de forma automatizada. Jenkins es compatible con el ciclo de vida completo del desarrollo de

software, desde la creación, las pruebas, la documentación del software, la implantación y otras etapas del ciclo de vida del desarrollo de software.



## Cómo crear un pipeline CI en Jenkins

### 1. Pipeline declarativo

El pipeline de Jenkins **se puede escribir de forma declarativa con la "sintaxis Jenkins"**. La sintaxis de Jenkins es un DSL (Lenguaje Específico de Dominio) Groovy que se utiliza para escribir las etapas del pipeline. El DSL se convierte internamente en XML para hacer el pipeline de Jenkins. Una "etapa" en un pipeline es un grupo de pasos a ejecutar en el pipeline.

Consideremos una etapa - "git checkout" que puede clonar el repositorio git (paso) y etiquetarlo con una versión (paso). Un pipeline con guión es fácil de crear y con la ayuda de la sintaxis de Jenkins, también se puede leer sin esfuerzo.

#### Ejemplo de canalización con guión de Jenkins

```
pipeline {
    agent any
    stages {
        stage('Git Checkout') {
            steps {
                echo 'Cloning repository'
                sh 'git clone https://github.com/digitalhouse.git'
            }
        }
        stage('Build') {
            steps {
                echo 'Building project'
                sh 'gradle clean build'
            }
        }
    }
}
```

### 2. Constructor de jobs Jenkins JJB (Jenkins job builder)

Es una **herramienta para crear una tubería utilizando la configuración YAML**. El pipeline de Jenkins puede ser escrito en YAML y con la ayuda de la herramienta JJB, convierte la configuración YAML en formato XML y lo envía a Jenkins para crear pipelines. Es comparativamente más fácil escribir la configuración en YAML que escribir el código completo en la sintaxis de Jenkins.

Esta herramienta también permite el uso de "Pipeline as Code" para que el código de un pipeline pueda ser enviado a un repositorio git y el pipeline sea creado o actualizado en Jenkins. La herramienta también ayuda a modelar el código de la tubería.

#### *Ejemplo del template*

```
- project:  
name: project-name  
pyver:  
- 26:  
branch_name: old_branch  
- 27:  
branch_name: new_branch  
jobs:  
- '{name}-{pyver}'
```

#### *Ejemplo de Job*

```
- job-template:  
name: '{name}-{pyver}'  
builders:  
- shell: 'git co {branch_name}'
```

La integración continua son construcciones automatizadas que pueden ser desencadenadas por algún tipo de evento, como el check-in del código, la fusión, o en un horario regular. El objetivo final de una compilación es ser desplegada en algún lugar, y el objetivo principal de la Integración Continua es construir y publicar esa unidad desplegable.

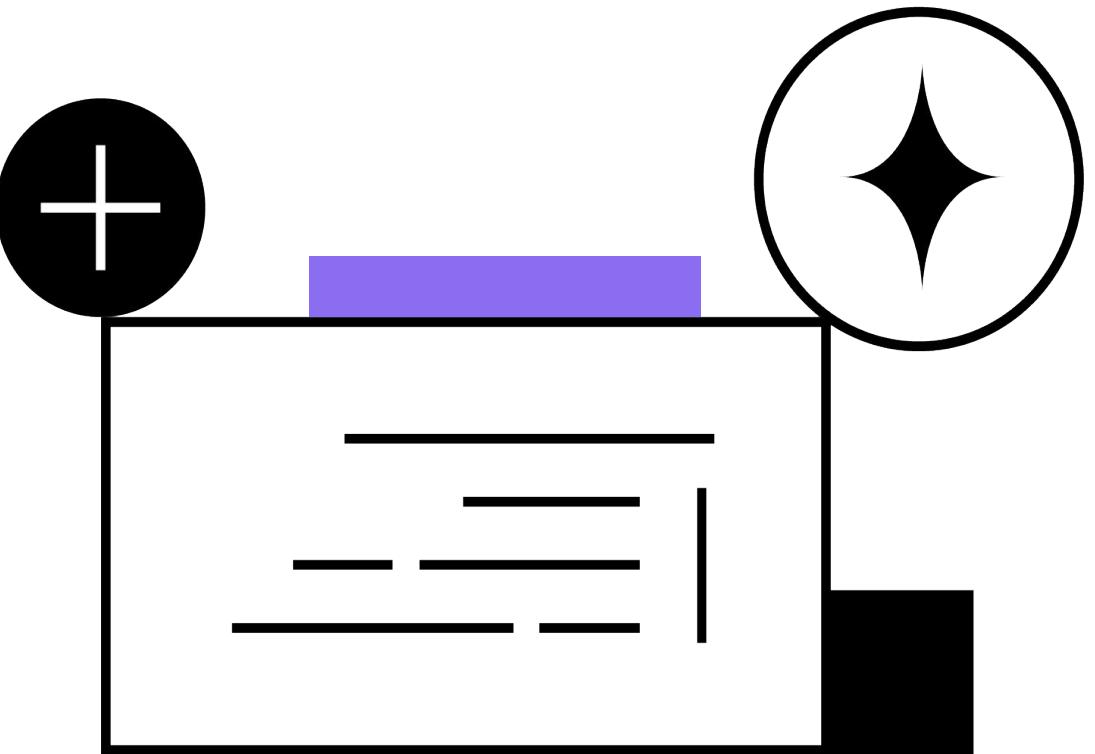
## Jenkins, pipelines y jobs

En resumidas palabras, hasta ahora lo que hicimos fue desarrollar pruebas automatizadas utilizando Selenium para FrontEnd y Rest Assured para BackEnd. Además aprendimos a crear suites de pruebas (conjuntos de pruebas) dentro de un proyecto de automatización. Es decir, que hasta ahora podemos ejecutar pruebas para una interfaz gráfica y para microservicios.

En esta clase vamos a profundizar sobre Jenkins y en cómo insertar un proyecto de automatización de pruebas en un proceso de integración continua. Así, dejaremos de ejecutar nuestras pruebas localmente (como lo hacíamos en Testing I o en el proyecto integrador) y nos prepararemos para correrlo en un servidor dedicado a tal fin.

# Variables de Entorno

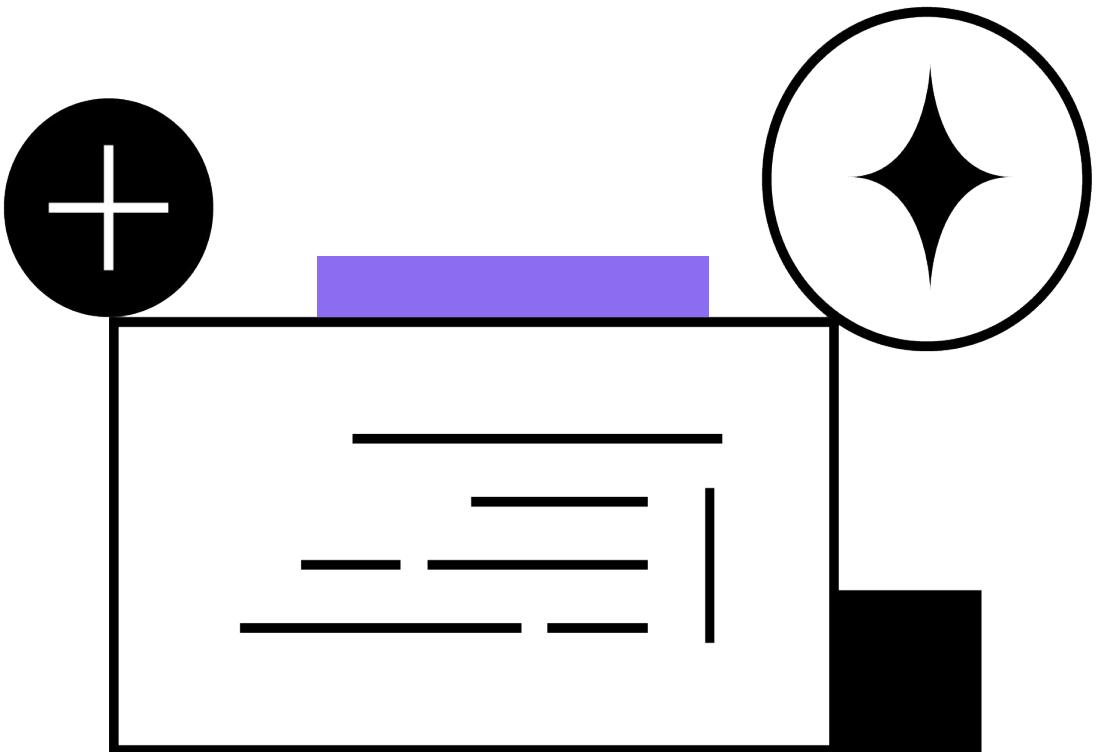
Jenkins nos permite usar las variables de entorno a través de la variable global **env**, que está disponible desde cualquier lugar dentro de un archivo **Jenkinsfile**.



# Variables de Entorno

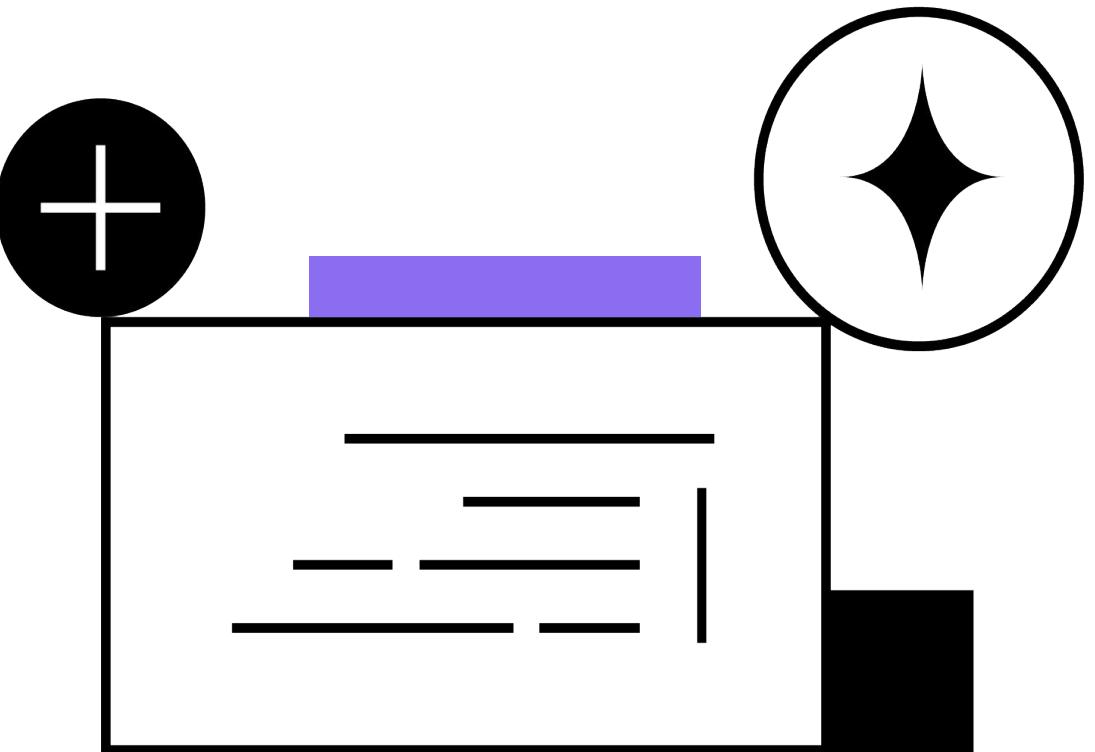
Algunas de estas variables son:

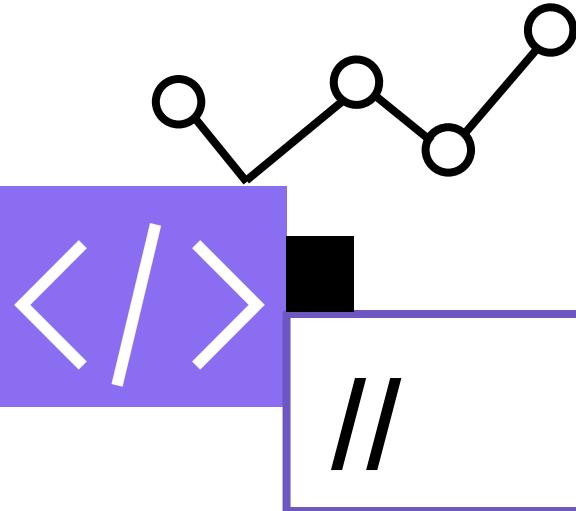
- **BUILD\_ID**: El ID de compilación actual, idéntico a BUILD\_NUMBER
- **BUILD\_NUMBER**: El número de compilación actual
- **BUILD\_TAG**: Cadena de jenkins-\${JOB\_NAME}-\${BUILD\_NUMBER}. Conveniente para poner en un archivo de recursos, un archivo jar, etc. para una identificación más fácil
- **BUILD\_URL**: La URL donde se pueden encontrar los resultados de esta compilación (por ejemplo, <http://buildserver/jenkins/job/MyJobName/17/>)



# Variables de Entorno

- **EJECUTOR\_NUMBER**: El número único que identifica al ejecutor actual (entre los ejecutores de la misma máquina) que realiza esta compilación. Este es el número que ve en el "estado del ejecutor de compilación", excepto que el número comienza desde 0, no desde 1
- **JAVA\_HOME**: Si su trabajo está configurado para usar un JDK específico, esta variable se establece en JAVA\_HOME del JDK especificado. Cuando se establece esta variable, PATH también se actualiza para incluir el subdirectorio bin de JAVA\_HOME
- **JENKINS\_URL**: URL completa de Jenkins, como <https://example.com:port/jenkins/> (Sólo disponible si la URL de Jenkins está configurada en "Configuración del sistema")



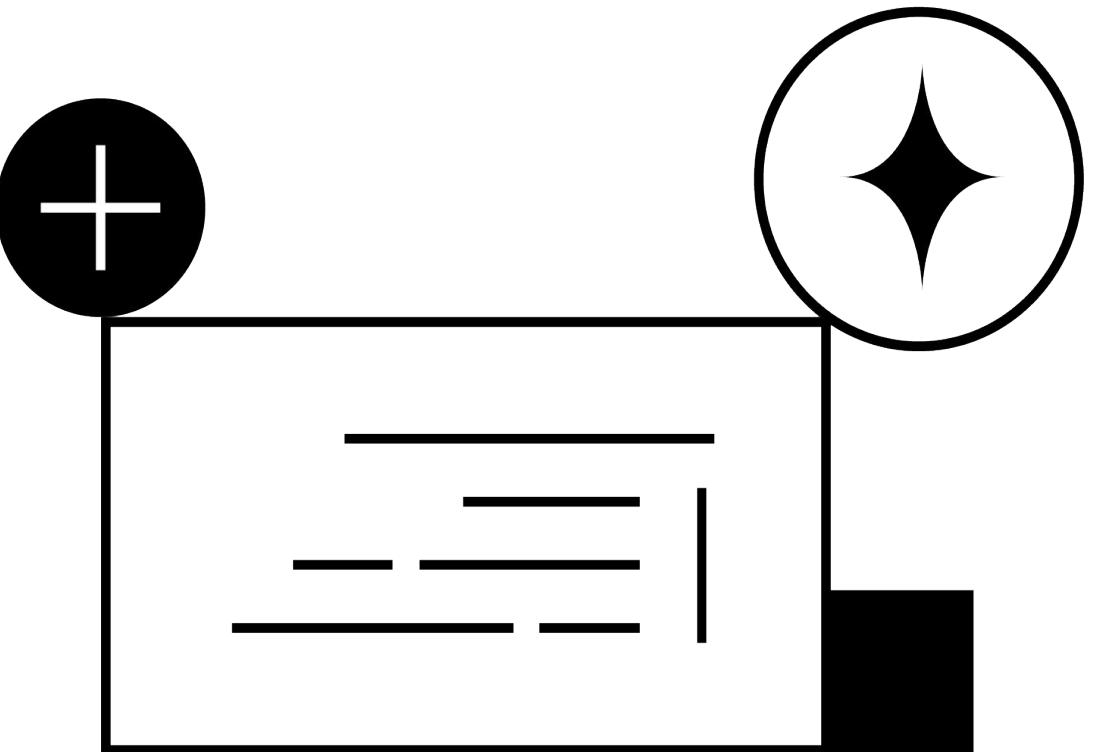


# Variables de Entorno

```
pipeline {  
    agent any  
    stages {  
        stage('Example') {  
            steps {  
                {}  
                echo "Running ${env.BUILD_ID} on ${env.JENKINS_URL}"  
            }  
        }  
    }  
}
```

# Setear Variables de Entorno

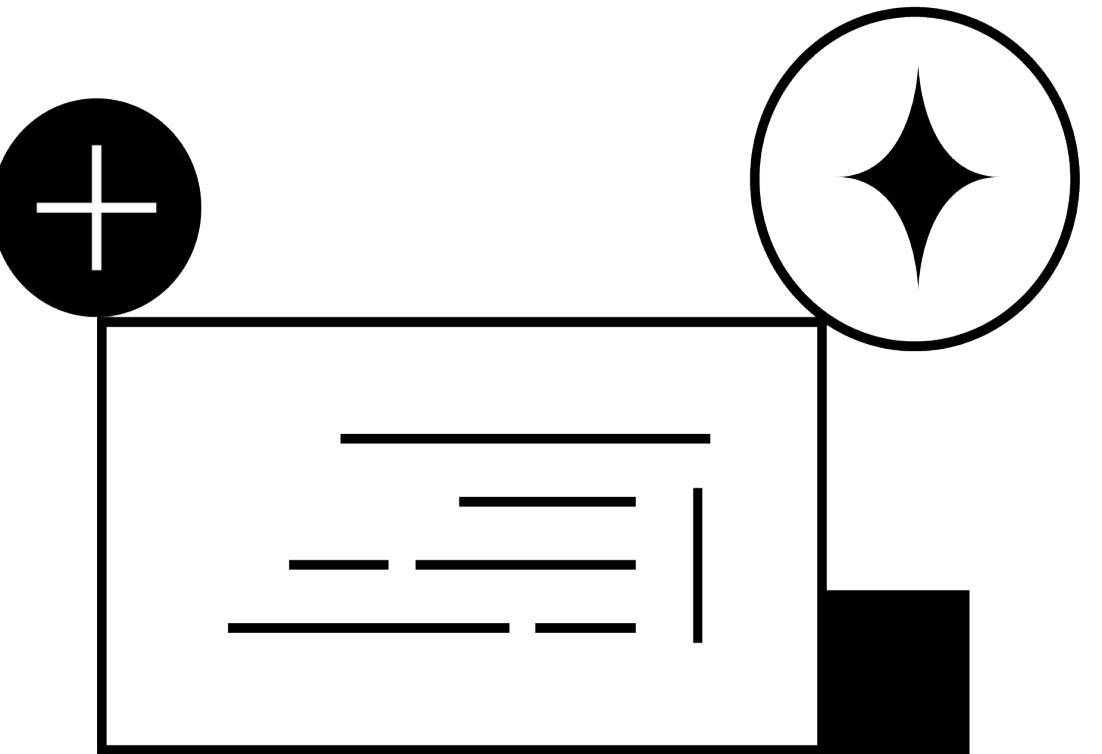
Jenkins nos permite setear nuestras propias variables de entorno, para ello, usamos [EnvInject](#), un plugin para injectar variables de entorno al inicio de la compilación.



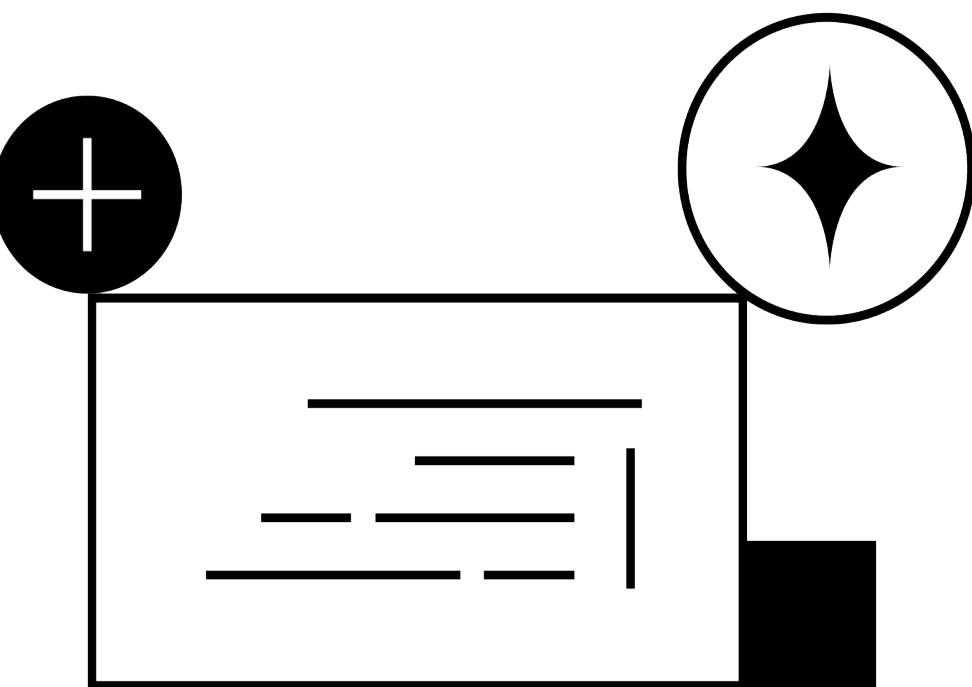
# Setear Variables de Entorno

Pasos:

1. Ir a la página de configuración del trabajo.
2. Localizar la sección Build Environment.
3. Activar la opción Inject environment variables to the build process (Injectar variables de entorno en el proceso de construcción).
4. En el campo Properties Content, modificar la variable de entorno que necesitamos.



# Setear Variables de Entorno



## Build Environment

- Add timestamps to the Console Output
- Generate environment variables from script
- Inject environment variables to the build process ?

Properties File Path



Properties Content

```
#extending path  
PATH=$PATH:/usr/local/bin  
  
#comment!  
FOO=BAR
```



```
-----  
[EnvInject] - Executing scripts and injecting environment variables after the SCM step.  
[EnvInject] - Injecting as environment variables the properties content  
PATH=/usr/local/bin:/usr/bin/python2.7:/home/jenkins/android-sdk-linux/tools:/home/jenkins/android-sdk-linux/platform-tools:/home/jenkins/arcanist/bin:/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games:/home/jenkins/android-sdk-linux/tools:/home/jenkins/android-sdk-linux/platform-tools:/usr/local/bin  
FOO=BAR
```

```
[EnvInject] - Variables injected successfully.
```



# Integración continua

CI y CD son términos comúnmente utilizados por los equipos de desarrollo.

Cuando hablamos de **integración continua (CI)** se trata de un proceso automatizado para integrar continuamente los cambios que se producen durante el desarrollo del software (construcción, prueba y validación del código). Con la ayuda de las funciones de CI se aceleran los ciclos de liberación del código evitando los conflictos de unión.

La IC requiere un sistema de control de versiones que pueda rastrear los cambios y las versiones del código del software. En este contexto, se suele utilizar Git. Así, los desarrolladores trabajan primero en su máquina local y luego fusionan su código con el repositorio compartido.

**La implementación de CI puede ser el primer paso para producir código de alta calidad en sus equipos,** debido a las validaciones. Hay muchas herramientas de integración continua disponibles, entre las más populares está Jenkins.

El producto final de la integración continua es una versión candidata, la forma final de un artefacto para ser desplegado. Pueden darse pasos de calidad para producir el artefacto, como la búsqueda de errores y la identificación de su solución. El empaquetado, la distribución y la configuración entran en una versión candidata. De esta manera, la IC alivia a los equipos de la carga de las construcciones manuales y también hace que las construcciones sean más repetibles, consistentes y disponibles

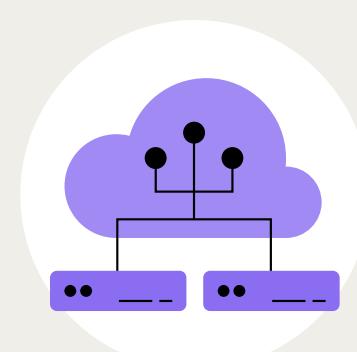
# Pilares de la CI

Las Pruebas de Integración Continua sirven como puerta de entrada a la calidad durante cada uno de los tres pilares de CI, construir, empaquetar y publicar artefactos.



## Construir

- Ejecutar herramientas de compilación dependientes del lenguaje, como en el ecosistema JAVA, como Maven/Gradle, o en el ecosistema NodeJS, como NPM. Su compilación combina su código escrito a medida con dependencias de terceros.



## Probar

- Como el artefacto aún no ha sido desplegado, las pruebas unitarias son excelentes para ejecutar en el proceso de CI. A medida que se construye el artefacto/unidad desplegable, las comprobaciones centradas en el artefacto, como las de código abierto y de dependencias, son esenciales para encontrar vulnerabilidades y posibles riesgos de licencia.



## Publicar

- La publicación en un repositorio/registro de artefactos se produce después de la creación de la unidad de artefacto/implante.

# Test de integración continua

Los tests de integración continua son aquellos enfocados y ejecutados durante el proceso de CI y orquestados por las herramientas de CI, las cuales, son responsables de construir, empaquetar y publicar artefactos.

Las pruebas en el proceso de IC **permiten una rápida retroalimentación** y, por diseño, **detienen la progresión del artefacto si no se cumple la calidad mínima**. Normalmente, las pruebas de CI se centran en el artefacto antes del despliegue en el primer entorno de integración.

Cuando el artefacto pasa las pruebas de integración continua, significa que se ha alcanzado un nivel de calidad antes de la liberación, no sólo que tiene la capacidad de ser construido de forma consistente.

Es importante señalar que una compilación fallida es mucho menos grave que un despliegue fallido y que es perfectamente normal que se encuentren errores y correcciones en las pruebas de CI.

# Test de IC

Estas pruebas se orquestan en una cadena y se dividen en dos categorías: pruebas dentro y fuera del proceso.

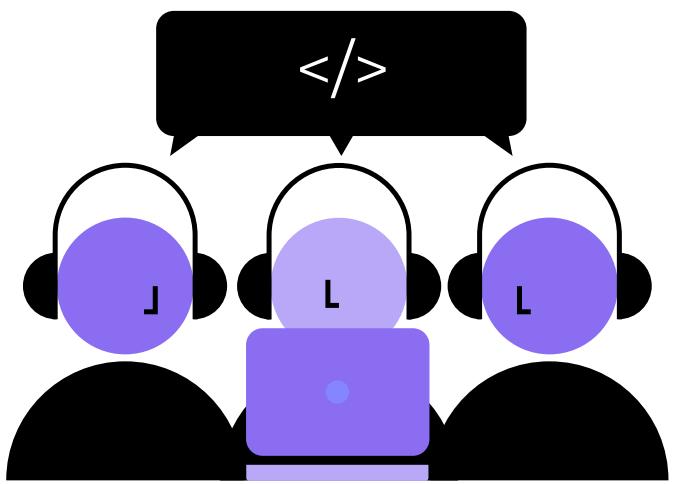
- **Test de calidad del código.** Las herramientas de calidad del código, como SonarQube y Checkmarx , se centran en el análisis estático del código y son prudentes durante los cambios de código.
- **Pruebas unitarias.** Las pruebas unitarias se centran en bloques/métodos de código que han cambiado, están diseñadas para ser granulares y se ejecutan juntas. Por lo general, se crean objetos simulados y se comprueban las afirmaciones.
- **Prueba de integración.** En el caso de las pruebas de integración en el contexto de la integración continua, se centrará en probar módulos cruzados de la aplicación. Las herramientas modernas de pruebas unitarias y de integración se solapan mucho. JUnit puede utilizarse para realizar pruebas unitarias y de integración, por ejemplo.
- **Prueba de seguridad/licencia.** El objetivo es encontrar la exposición y el riesgo de utilizar determinados paquetes. Las herramientas de análisis de seguridad/licencias como Blackduck , Snyk y StackHawk tienen diferentes métodos de introspección: algunos requieren que se ejecute el artefacto terminado, por ejemplo, una imagen docker.



# Jenkins

Jenkins es una **solución de código abierto** que cuenta con el apoyo de proveedores, principalmente Cloudbees.

Jenkins es un servidor de Integración Continua de código abierto escrito en Java para orquestar una cadena de acciones para lograr el proceso de Integración Continua de forma automatizada, soportando el ciclo de vida completo del desarrollo de software.



# Pipeline

Un pipeline es una serie de pasos orquestados con la capacidad de llevar el código fuente a producción. Las etapas incluyen:

- Construcción
- Envasado
- Pruebas
- Validación
- Verificación de la estructura
- Despliegue

Dependiendo de las estructuras organizativas y de los equipos, se requiere el uso de varios conductos para lograr este objetivo.

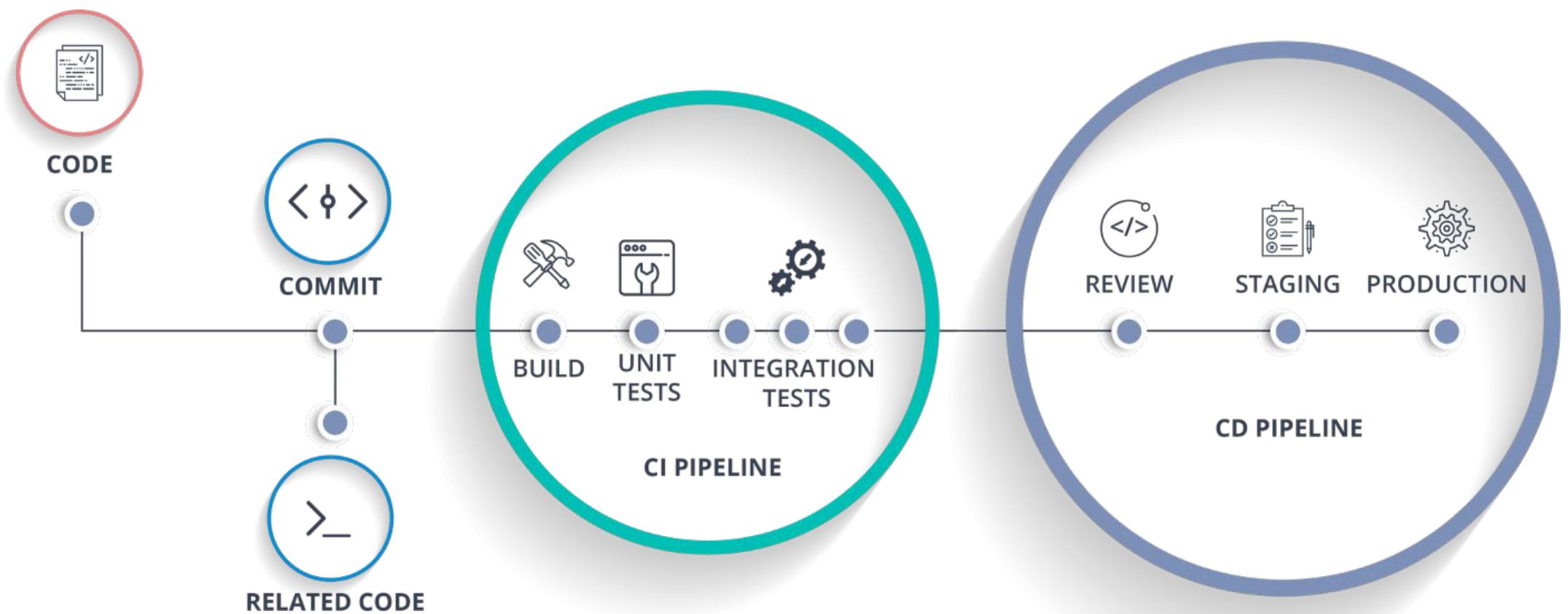
Pipeline predica la continuidad, es decir, **incluye la integración continua, la entrega/despliegue continuo (CI/CD), el feedback continuo y las operaciones continuas**. Así, en lugar de pruebas puntuales o despliegues programados, cada función se produce continuamente.



# Jobs

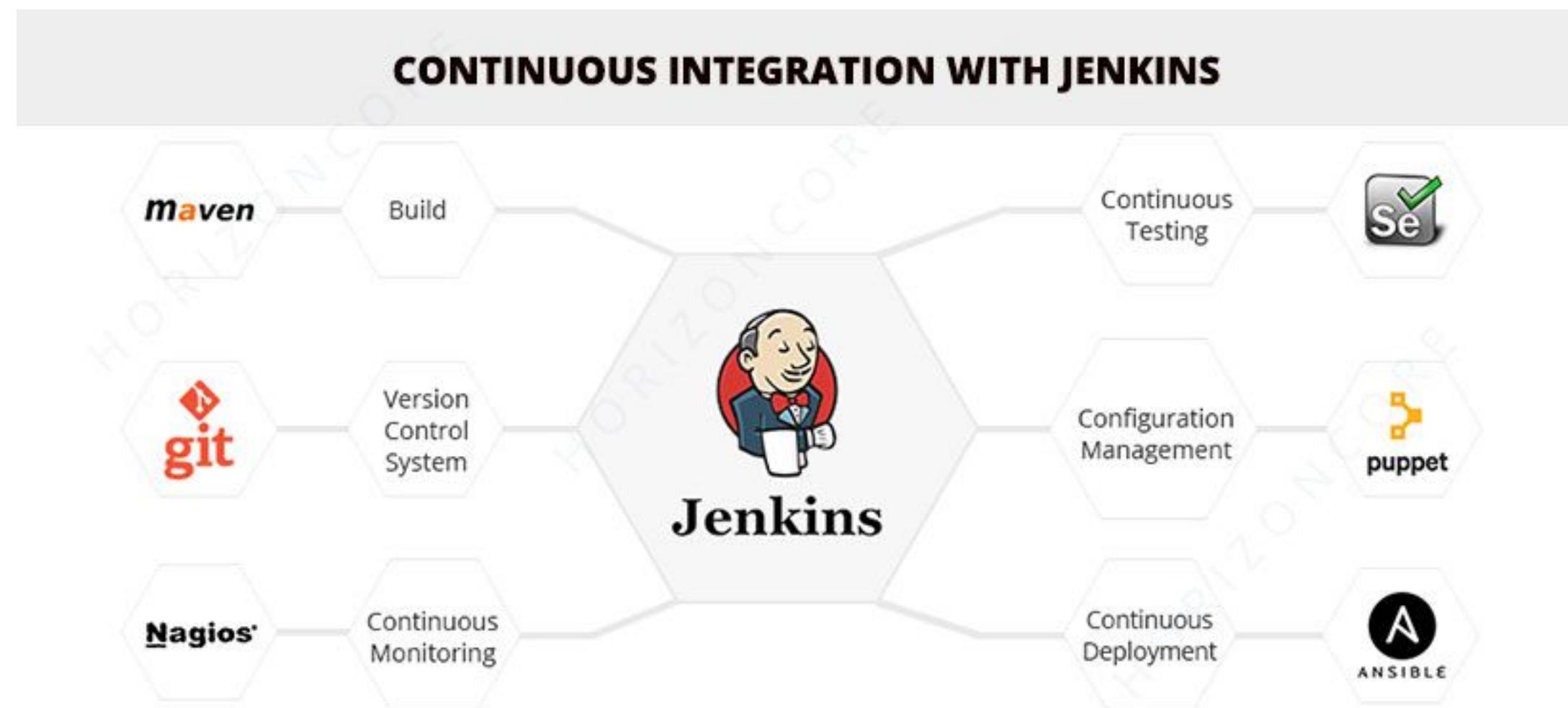
El trabajo se refiere a cada tarea dentro de la tubería, estas tareas pueden ser interdependientes y hacer llamadas cuando se empuja, a través de la programación o manualmente. Construir es cada una de las ejecuciones de Job.

El CD de flujo completo de CI:



# Jobs

Ver la relación entre las herramientas de CI y las actividades:



# Etapas del proceso de control de calidad

Ya vimos las diferencias entre Plan de Pruebas Maestro (Master Test Plan) y el Plan de Pruebas de Lanzamiento (Release Test Plan).

Es momento de conocer las etapas del proceso de control de calidad, ¿cuáles son? ¿Qué funciones implican? ¿Para qué sirven?



## Anализar Requisitos

Cuesta más arreglar un fallo detectado durante las pruebas que prevenirlo en la fase de diseño de los requisitos. Los profesionales del QA deben participar en el análisis y la definición de los requisitos del software, tanto funcionales como no funcionales. Los QAs deben recibir requisitos coherentes, exhaustivos, trazables y claramente marcados. Esto ayuda al equipo de QA a diseñar pruebas específicamente adaptadas al software que se está probando.

## Planear las pruebas

La información obtenida durante la fase de análisis de requisitos se utiliza como base para planificar las pruebas necesarias. El plan de pruebas debe incluir la estrategia de pruebas de software, el alcance de las pruebas, el presupuesto del proyecto y los plazos establecidos. También debe describir los tipos y niveles de pruebas necesarios, los

métodos y herramientas para el seguimiento de los errores, y asignar recursos y responsabilidades a cada uno de los probadores.

### Diseñar las pruebas

En esta fase, los equipos de control de calidad deben crear casos de prueba y listas de comprobación que cubran los requisitos del software. Cada caso de prueba debe contener condiciones, datos y los pasos necesarios para validar cada funcionalidad. Cada test debe definir también el resultado esperado de la prueba para que los probadores sepan con qué comparar los resultados reales.

01

Pasos para construir un  
plan maestro de  
pruebas

# PASOS

01

Análisis del producto

02

Diseño de la estrategia de pruebas

03

Fijación de objetivos

04

Establecer los criterios de prueba

05

Planificación de la asignación de recursos

06

Planificación de la configuración del entorno de pruebas

07

Determinación del calendario de pruebas y estimación

# 1. Análisis del producto

Empecemos por **aprender más sobre el producto** que se está probando, el cliente y los usuarios finales de productos similares.

Lo ideal es que esta fase se centre en responder a las siguientes preguntas:

- ¿Quién utilizará el producto?
- ¿Cuál es el objetivo principal de este producto?
- ¿Cómo funciona el producto?
- ¿Cuáles son las especificaciones de software y hardware?

**En esta fase, les recomendamos:**

- Entrevistar a clientes, diseñadores y desarrolladores.
- Revisar la documentación del producto y del proyecto.
- Realizar un recorrido por el producto.

## 2. Diseño de la estrategia de pruebas

El documento de estrategia de pruebas lo elabora el director de pruebas y define lo siguiente:

- Objetivos del proyecto y cómo alcanzarlos.
- La cantidad de esfuerzo y coste que requiere la prueba.

## 2. Diseño de la estrategia de pruebas

Más concretamente, el documento debe detallar:

- **Alcance de la prueba:** contiene los componentes de software (hardware, software) que se van a probar y también los que no se van a probar.
- **Tipo de prueba:** describe los tipos de pruebas que se utilizarán en el proyecto. Esto es necesario porque cada prueba identifica tipos específicos de errores.
- **Riesgos y problemas:** describe todos los posibles riesgos que pueden producirse durante las pruebas —plazos ajustados, gestión insuficiente, estimaciones presupuestarias inadecuadas o erróneas —así como el efecto de estos riesgos sobre el producto o la empresa.
- **Logística de las pruebas:** menciona los nombres de los testers así como las pruebas que deben realizar. Esta sección también incluye las herramientas y el calendario establecido para las pruebas.



### 3. Definición de objetivos

En esta fase se definen **los objetivos y los resultados esperados** de la ejecución de la prueba. Como todas las pruebas tienen como objetivo identificar el mayor número posible de defectos, los objetos deben incluir:

- Una lista de todas las características del software —funcionalidad, GUI, normas de rendimiento— que deben probarse
- El resultado ideal o punto de referencia para cada aspecto del software que necesita ser probado. Este es el punto de referencia contra el cual se compararán todos los resultados reales.

## 4. Establecer los criterios de prueba

Los criterios de prueba se refieren a las **normas o reglas que rigen todas las actividades de un proyecto de prueba**. Los dos criterios principales de la prueba son:

- **Criterios de suspensión:** establece los **puntos de referencia para suspender todas las pruebas**. Por ejemplo, si los miembros del equipo de control de calidad descubren que el 50% de los casos de prueba han fallado, se suspenderán todas las pruebas hasta que los desarrolladores resuelvan todos los errores identificados hasta el momento.
- **Criterios de salida:** define los puntos de referencia que significan la finalización con éxito de una fase de prueba o proyecto. Los criterios de salida son los **resultados esperados de las pruebas** y deben cumplirse antes de pasar a la siguiente fase de desarrollo. Por ejemplo, el 80% de todos los casos de prueba deben ser calificados como exitosos antes de que una característica particular o pieza de software pueda ser considerada adecuada para el uso público.



# 5. Planificación de la asignación de recursos

Esta fase crea un desglose detallado de todos los recursos necesarios para completar el proyecto. Los recursos incluyen el esfuerzo humano, el equipo y toda la infraestructura necesaria para realizar pruebas precisas y completas.

Esta parte del plan de pruebas **decide la medida de los recursos**—número de probadores y equipos— que requiere el proyecto. También **ayuda a los responsables de las pruebas** a formular un calendario y una estimación correctamente calculados para el proyecto.

# 6. Planificación de la configuración del entorno de pruebas

El entorno de pruebas se refiere a la configuración de software y hardware en la que los QAs realizan sus pruebas. **Lo ideal es que estos entornos de prueba sean independientes** y solo utilizado por el equipo de calidad. Además, tecnológicamente, debería ser lo más parecido posible al entorno productivo.

# 7. Determinación y estimación del calendario de pruebas

Para la estimación de las pruebas, dividan el proyecto en tareas más pequeñas y asignen el tiempo y el esfuerzo necesarios para cada una. Seguramente, en este paso se utilicen técnicas de estimación de esfuerzo.

A continuación, conviene crear un calendario para completar estas tareas en el tiempo designado con la cantidad específica de esfuerzo.

Sin embargo, la creación del calendario requiere la participación de varias perspectivas:

- Disponibilidad de personal, número de días de trabajo, plazos de los proyectos, disponibilidad diaria de recursos.
- Riesgos asociados al proyecto que han sido evaluados en una fase anterior.

## Testing II

# Procedimientos de prueba en las automatizaciones

Se requiere un conjunto diferente de productos antes, durante y después de la prueba:

## Etapa 1: Definir el alcance de la automatización

El **alcance** de la automatización significa el área de su Aplicación bajo Prueba que será automatizada. Asegúrense de haber recorrido con precisión y conocer el estado de las pruebas de tu equipo, la cantidad de datos de prueba y también el entorno en donde se realizan las pruebas.

A continuación se ofrecen consejos adicionales para ayudarte a determinar el alcance de las pruebas automatizadas:

- Viabilidad técnica
- La complejidad de los casos de prueba
- Las características o funciones que son importantes para el negocio
- El grado de reutilización de los componentes del negocio
- La capacidad de utilizar los mismos casos de prueba para las pruebas entre navegadores.

## Etapa 2: Seleccionando una herramienta de prueba

Después de determinar su alcance, es el momento de **elegir una herramienta para las pruebas de automatización**. Por supuesto, pueden seleccionar entre una amplia gama de herramientas de automatización disponibles en el mercado. Sin embargo, sólo depende de la tecnología sobre la que se construyen las pruebas de las aplicaciones. Cada tipo de herramienta o marco de trabajo puede satisfacer diferentes demandas, por lo que tener un conocimiento profundo de los distintos tipos de herramientas es también un factor destacado para elegir la mejor herramienta.

## Etapa 3: Planificación, diseño y desarrollo

En esta fase, crearán una estrategia y un plan de automatización. Este plan puede incluir los siguientes elementos:

- La herramienta de prueba de automatización que se haya seleccionado
- El diseño del framework y sus características
- Un cronograma detallado para el scripting y la ejecución de los casos de prueba
- Elementos de automatización dentro y fuera del ámbito de aplicación
- Objetivos y resultados del proceso de pruebas de automatización

## Etapa 4: Ejecutar casos de prueba y crear los informes

Una vez que hayan completado todos los pasos anteriores, **¡es el momento de actuar!** Pueden escribir los scripts, ejecutar las pruebas automáticamente ejecutando el código directamente o llamando a la API o a la interfaz de usuario de una aplicación. Tras su ejecución, el informe de pruebas ofrece un resumen consolidado de las pruebas realizadas hasta el momento para el proyecto.

## **Etapa 5: mantenimiento de los casos de prueba anteriores**

No importa lo bien que gestionen sus pruebas de automatización, el mantenimiento de las pruebas es inevitable si quieren ampliar su colección de scripts de prueba reutilizables. Después de que las pruebas automatizadas se hayan programado y ejecutado, es necesario actualizarlas si la aplicación cambia la próxima vez. Recuerden el principio de testing '**La paradoja del pesticida**', si sus pruebas no van adaptándose al sistema bajo prueba, quedarán obsoletas.

## Testing II

# Métricas

Podemos definir las métricas como una interpretación de una medida (cuantificable) o un conjunto de medidas utilizadas para analizar el resultado de un proceso, acción o estrategia específicos.

Las métricas, en la calidad del software, pueden recogerse durante y al final de las actividades de prueba, y pueden tener diferentes propósitos. Así, se facilita la comunicación sobre el progreso de las pruebas.

Pueden clasificarse en las siguientes categorías de métricas:

- Métricas de proyecto: medir el progreso con respecto a los criterios de resultados esperados del proyecto, como el porcentaje de casos de prueba ejecutados, aprobados y fallados.
- Métricas de producto: medir algunos atributos del producto, como el grado de comprobación o la densidad de defectos.
- Métricas de proceso: medir la capacidad del proceso de pruebas de desarrollo, como el porcentaje de defectos encontrados por la prueba.
- Métricas de personas: medir la capacidad de individuos o grupos, como la realización de casos de prueba en un plazo determinado.

Hay cinco dimensiones principales en las que se controla el progreso de la prueba:

- Riesgos (de calidad) del producto
- Defectos
- Pruebas
- Cobertura
- Confianza

Entre las posibles métricas están:

- Porcentaje de trabajo planificado realizado en la preparación de casos de prueba —o porcentaje de casos de prueba planificados implementados—.
- Porcentaje de trabajo planificado realizado en la preparación del ambiente de pruebas.

- Ejecución de casos de prueba —por ejemplo, número de casos de prueba ejecutados/no ejecutados, casos de prueba superados/no superados o condiciones de prueba superadas/no superadas—.
- Información sobre defectos —por ejemplo, densidad de defectos, defectos encontrados y corregidos, tasa de fallos y resultados de pruebas de confirmación—.
- Cobertura de pruebas de requisitos, historias de usuario, criterios de aceptación, riesgos o código.
- Realización de tareas, asignación y utilización de recursos y esfuerzos.
- Coste de las pruebas, incluyendo el coste comparado con el beneficio de encontrar el siguiente defecto o el coste comparado con el beneficio de realizar la siguiente prueba.

# ¿Para qué medir?

Antes de definir qué medir, es importante tener claro por qué medir la calidad de la automatización de pruebas, dado que esta práctica puede ser diferente en cada empresa. Algunas de las razones pueden ser para evaluar:

- Eficacia de la automatización
- Velocidad de entrega (debido a la automatización)
- Mejora continua
- Cultura de responsabilidad

# Métricas de pruebas automatizadas

En general, la mayoría de las métricas de pruebas automatizadas evalúan:

- ➔ **Cobertura:** medir el alcance de las pruebas con respecto al software en desarrollo
- ➔ **Progreso:** medir lo que se puede mejorar con el tiempo (por ejemplo, el tiempo para corregir un defecto).
- ➔ **Calidad:** medida de prueba que indica la calidad del software, por ejemplo, el rendimiento.

# Casos de prueba automatizables



## Definición

Esta métrica puede ayudarle a comprender dónde los equipos están dando prioridad a la automatización o qué áreas pueden seguir requiriendo validaciones manuales. Desglosar esta métrica para cubrir componentes específicos de su aplicación puede proporcionar un valor aún mayor.



## ¿Cómo calcular?

$\% \text{ Automatizable} = (\text{Cantidad de pruebas automatizables} / \text{Cantidad de pruebas totales}) * 100$

# Eficacia del script de automatización



## Definición

Esta métrica proporciona claridad sobre **cómo se están encontrando los defectos**. Si estás invirtiendo mucho dinero en la automatización, pero tus scripts de automatización no están encontrando defectos, es posible que quieras investigar y entender la eficacia de esos scripts. Además, si utilizan diferentes entornos de prueba, como el de integración y el de puesta en escena, pueden esperar una menor efectividad en el entorno de integración porque el script no se ha completado para las nuevas características. Del mismo modo, en el entorno de pruebas, es de esperar que la automatización encuentre la mayoría de los defectos, ya que normalmente se centra en las pruebas de regresión.



## ¿Cómo calcular?

Eficacia del script de automatización = (número de defectos encontrados por la automatización / número de defectos abiertos) \* 100

# Tasa de aprobación de la automatización



## Definición

Esta métrica proporcionará un porcentaje estándar de **cuántas pruebas de automatización se han superado**. Es útil para conocer la estabilidad de la suite de la automatización, así como su eficacia.

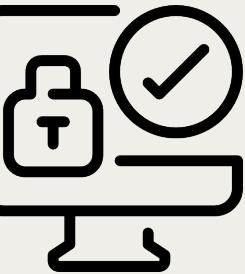
Tener un bajo índice de aprobados requiere dedicar más tiempo a la validación de los fallos. Si los fallos son falsos, esto es un indicador temprano de que la suite de automatización no es fiable. Además, si ven que este número disminuye después de una ejecución de automatización, esto puede ser un indicador rápido de que la versión contiene un número de defectos más alto de lo normal.



## ¿Cómo calcular?

Porcentaje de aprobación = (número de casos aprobados / número de casos de prueba realizados) \* 100

# Tiempo de ejecución de la automatización



## Definición

La métrica proporcionará un valor de **cuánto tiempo tarda su conjunto de automatización en ejecutarse de principio a fin**. Un conjunto de automatización de larga duración no es valioso cuando se trata de proporcionar retroalimentación a los desarrolladores o entregar el código a la producción rápidamente.



## ¿Cómo calcular?

Tiempo de ejecución = Tiempo Final – Tiempo Inicial

# Cobertura de las pruebas de automatización



## Definición

La medición de la cobertura de las pruebas ayudará a un equipo a entender **cuánta cobertura está proporcionando la suite de automatización frente a cuántas pruebas manuales se están realizando**. Se puede hacer un seguimiento por componentes y luego resumir los datos para visualizarlos en un conjunto de regresión.



## ¿Cómo calcular?

Cobertura de las pruebas de automatización =  
(número de pruebas de automatización / número de pruebas totales) \* 100

# Estabilidad de la automatización



## Definición

Esta métrica les **ayudará a evaluar el rendimiento de la automatización a lo largo del tiempo**. Tus scripts pueden fallar si algo en el sistema cambia y la automatización no se actualiza antes de la ejecución. Sin embargo, si con el tiempo, las pruebas fallan continuamente, esto es un buen indicador de que las mismas pueden no ser estables.



## ¿Cómo calcular?

Estabilidad de la Automatización = por prueba  
(Cantidad de fallos/Cantidad de ejecuciones) \*  
100

# Estabilidad de la compilación



## Definición

Si la automatización se encuentra en un pipeline CI/CD, pueden utilizar esta métrica para entender el nivel de calidad del código que proviene del desarrollo. Si sus compilaciones se rompen constantemente, puede haber una oportunidad para mejorar las pruebas unitarias, por ejemplo.



## ¿Cómo calcular?

$$\% \text{ de estabilidad de construcción} = (\text{número de fallos de construcción} / \text{número de construcciones}) * 100$$

# Automatización de Sprint



## Definición

La métrica ayuda a **identificar lo cerca que se está de la automatización en el sprint**. Para ser más valioso, el equipo debe esforzarse por automatizar el nuevo trabajo en la iteración en la que se completa el trabajo, no en un sprint posterior.



## ¿Cómo calcular?

Automatización en el sprint % = (Cantidad de scripts creados en el sprint / Cantidad de scripts creados después del sprint) \* 100

# Progreso de la automatización



## Definición

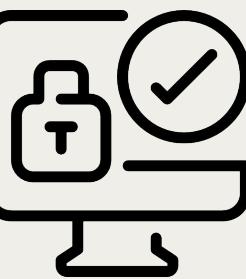
Independientemente de si están midiendo esto con respecto al objetivo de automatizar un conjunto de regresión o una nueva característica, esta métrica les ayudará a determinar cómo están progresando hacia ese objetivo en el tiempo.



## ¿Cómo calcular?

Progreso de la automatización% = (número de pruebas automatizadas / número de pruebas automatizables) \* 100

# Pirámide de automatización



## Definición

Aunque hay varias teorías sobre cómo debería ser la pirámide de automatización, el resultado es un entendimiento general de que una cierta cantidad de pruebas debería estar en los niveles de pruebas. Es importante disponer de métricas para controlar continuamente si la cobertura de la automatización de las pruebas sigue la estrategia de la pirámide.

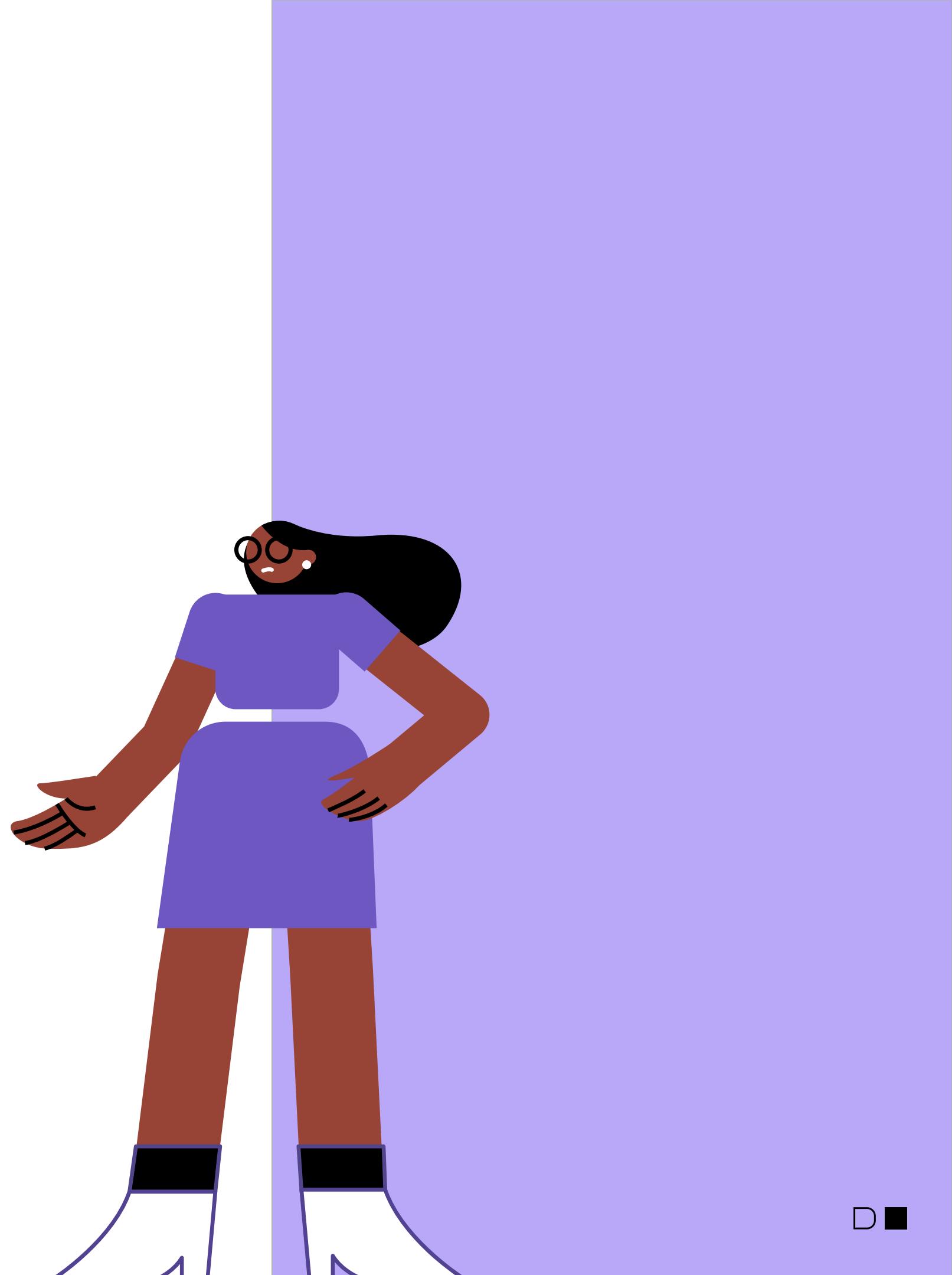


## ¿Cómo calcular?

Pirámide de automatización = (número de pruebas en cada nivel / número de pruebas totales) \* 100

# Conclusiones

Las **métricas de automatización de pruebas** correctas deben proporcionar una **comprensión objetiva y profunda del sistema**, el proceso y las pruebas de software, con un enfoque en la **identificación y solución de problemas** al tiempo que aumenta la eficiencia y la productividad de los equipos.



## Métricas relacionadas al proceso

### Productividad en la preparación de casos de prueba

#### **Descripción:**

Se utiliza para calcular el número de casos de prueba preparados y el esfuerzo invertido en la preparación de los casos de prueba.

#### **¿Cómo calcular?**

Productividad en la preparación de casos de prueba = Número de casos de prueba / Esfuerzo invertido en la preparación de casos de prueba (en horas).

### Cobertura del proyecto de prueba

#### **Descripción:**

Ayuda a medir el porcentaje de cobertura de los casos de prueba en relación con el número de requisitos.

#### **¿Cómo calcular?**

Cobertura del proyecto de prueba = Número total de requisitos asignados a los casos de prueba / Número total de requisitos.

### Productividad de ejecución de pruebas

#### **Descripción:**

Ayuda a medir el porcentaje de cobertura de los casos de prueba en relación con el número de requisitos.

#### **¿Cómo calcular?**

Cobertura del proyecto de prueba = Número total de requisitos asignados a los casos de prueba / Número total de requisitos.

### Cobertura de ejecución de pruebas

#### **Descripción:**

Se trata de medir el número de casos de prueba ejecutados en relación al número de casos de prueba previstos.

#### **¿Cómo calcular?**

Cobertura de la ejecución de la prueba = Número total de casos de prueba ejecutados / Número total de casos de prueba previstos para la ejecución.

## Casos de prueba aprobados, fallidos y bloqueados

### Descripción:

Se trata de medir el porcentaje de casos de prueba aprobados, fallidos y bloqueados.

### ¿Cómo calcular?

Casos de prueba superados = Número total de casos de prueba aprobados / Número total de casos de prueba ejecutados.

**Casos de prueba fallidos** = Número. Número total de casos de prueba fallidos / Número total de casos de prueba realizados.

**Casos de prueba bloqueados** = Número total de casos de prueba bloqueados/ Número total de casos de prueba ejecutados.

## Métricas relacionadas con el producto

## Tasa de detección de errores

### Descripción:

Es para determinar la eficacia de los casos de prueba.

### ¿Cómo calcular?

Tasa de detección de errores = (Número total de defectos encontrados / Número total de casos de prueba realizados) \*100.

## Tasa de corrección de defectos

### Descripción:

Ayuda a conocer la calidad del software en términos de reparación de defectos.

### ¿Cómo calcular?

Tasa de corrección de defectos = [(Nº total de defectos notificados como corregidos - Nº total de defectos reabiertos) / (Nº total de defectos notificados como corregidos + Nº total de nuevos errores debidos a la corrección)]. \* 100.

## Densidad de defectos

### Descripción:

Se define como la relación entre los defectos y los requisitos.

### ¿Cómo calcular?

Densidad de defectos = Número total de defectos identificados / Tamaño real (número de requisitos).

## Fugas por defecto

### Descripción:

Se utiliza para comprobar la eficacia del proceso de pruebas antes de la UAT (prueba de aceptación del usuario).

### ¿Cómo calcular?

Fuga por defecto = ( $N^{\circ}$  de defectos encontrados en UAT /  $N^{\circ}$  de defectos encontrados antes de la UAT) \*100.

## Eficacia de la eliminación de defectos

### Descripción:

Permite comparar la eficacia global de la eliminación de defectos —defectos encontrados antes y después de la entrega—.

### ¿Cómo calcular?

Eficiencia de la eliminación de defectos = [ $N^{\circ}$  total de defectos encontrados antes de la entrega / ( $N^{\circ}$  total de defectos encontrados antes de la entrega +  $N^{\circ}$  total de defectos encontrados después de la entrega)]. \*100.

## Reportes

La información recopilada en el seguimiento de las pruebas se consolida en reportes emitidos periódicamente a los implicados, para comunicar el progreso de la prueba. Pueden contener la siguiente información:

- El estado de las actividades de prueba y el progreso con respecto al plan de pruebas.
- Factores que impiden el progreso.
- Pruebas previstas para el próximo período de notificación.
- La calidad del objeto de ensayo.

Por otro lado, se emite un reporte resumido cuando se cumplen los criterios de salida. Esto puede incluir:

- Resumen de la prueba realizada.
- Información sobre lo ocurrido durante un periodo de prueba.

- Las desviaciones del plan incluyen cambios en el calendario, la duración o el esfuerzo de las actividades de prueba.
- Estado de la prueba y calidad del producto con respecto a los criterios de salida.
- Factores que han bloqueado o siguen bloqueando el progreso.
- Métricas para los defectos, los casos de prueba, la cobertura de las pruebas, el progreso de las actividades y el consumo de recursos.
- Riesgos residuales.
- Desarrollo de productos de trabajo de prueba reutilizables.

Cabe mencionar que el contenido específico de los reportes de pruebas varía en función del contexto del proyecto y del público al que va dirigido el reporte de pruebas, es decir, el tipo y la cantidad de información que se incluye para un público técnico o un equipo de pruebas puede ser diferente de la que se añade al reporte de resumen ejecutivo.

En el desarrollo con metodologías ágiles, el reporte de progreso de las pruebas puede incorporarse a los tableros de tareas, a los resúmenes de defectos y a los gráficos de trabajo pendiente, que pueden ser objeto de discusión durante las ceremonias de esta metodología.

En resumen, las métricas son importantes para evaluar el progreso de las pruebas, ya sean manuales o automatizadas. Deben definirse para facilitar la comunicación, la retroalimentación y la visibilidad del proceso de pruebas. Además, podemos definir métricas de proyecto, producto, proceso y personas y los cálculos son bastante sencillos.

Es importante identificar el porqué de cada métrica, para que realmente contribuya al equipo y al proceso de desarrollo de software, especialmente cuando hablamos de medir las pruebas automatizadas. Además, es válido certificar que las métricas definidas están bien comunicadas, especialmente su importancia.

Una forma de proporcionar las métricas es a través de reportes, que pueden generarse en dos momentos: durante y al final del ciclo de pruebas.

# Revisión: segunda parte

# Índice

- 01** [Armado de suites](#)
- 02** [Reportes](#)
- 03** [REST Assured](#)
- 04** [Integración continua](#)
- 05** [Jenkins, pipelines y jobs](#)
- 06** [Master test plan y release test plan](#)
- 07** [Métricas y reportes](#)



01

# Armado de suites

# Armado de suites, playlist y conjuntos de tests dentro de nuestro proyecto

Dada la necesidad de agrupar los casos de prueba para su ejecución en un ciclo específico, los equipos pueden crear lo que en calidad de software llamamos **suites de prueba**.

Comúnmente, se crea la smoke suite que reúne los casos de prueba principales, los esenciales que deben ser ejecutados antes de la liberación de la aplicación al usuario final (utilizada también en casos de emergencia) y la suite de regresión, que asegura la ejecución del grupo de casos de prueba.

Este comprueba el funcionamiento de la aplicación después de un determinado cambio, para encontrar posibles daños colaterales de los cambios.

En el framework presentado en el curso, podemos agrupar los casos de prueba en suites mediante el uso de la anotación **@Tag**.

02

## Reportes

**Los informes/reportes sirven para consolidar la información detallada sobre el resultado de los casos de prueba y dan visibilidad de la calidad del software a las partes interesadas.**

En general, los informes contienen la cantidad de pruebas ejecutadas, la cantidad de pruebas superadas, la cantidad de pruebas fallidas y los detalles respectivos de las pruebas. También pueden incluir capturas de pantalla.



# Generación de reportes desde nuestro proyecto de automatización

Son importantes porque:

- Ayudan a visualizar y resumir los resultados de las pruebas.
- Desempeñan un papel fundamental en la automatización de las pruebas.
- Ayudan a los testers y a otras partes interesadas a comprender la estabilidad de las pruebas realizadas antes de que el producto salga a la luz, lo que proporciona confianza para la puesta en marcha y la posterior puesta en producción.

# Generación de reportes

- En el marco utilizado durante el curso, disponemos de la biblioteca de código abierto Extent Report para generar estos informes en formato HTML. En la práctica debemos:

1- Añadir la dependencia en el archivo **POM.xml**:

```
<!-- https://mvnrepository.com/artifact/com.relevantcodes/extentreports -->
<dependency>
    <groupId>com.relevantcodes</groupId>
    <artifactId>extentreports</artifactId>
    <version>2.41.2</version>
</dependency>
```

# Generación de reportes

2- A continuación, instalar en el proyecto.

```
ExtentReports extent = new ExtentReports();
ExtentSparkReporter spark = new ExtentSparkReporter("target/Spark.html");
extent.attachReporter(spark);
extent.createTest("MyFirstTest")
.log(Status.PASS, "This is a logging event for MyFirstTest, and it passed!");
extent.flush();
```

03

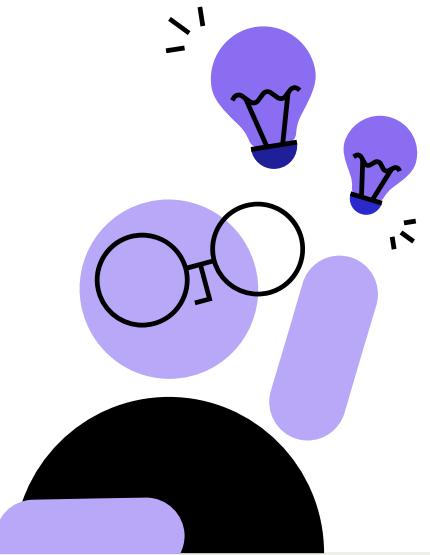
## REST Assured

# Introducción al testing de microservicios con REST Assured y Java

Ahora que las API y los microservicios desempeñan un papel cada vez más importante en el desarrollo de software, las pruebas automatizadas de estas API y microservicios se están volviendo indispensables. Así que hablemos de una de las herramientas de código abierto más populares para esta tarea: **REST Assured**. Esta es una biblioteca Java que proporciona un lenguaje específico de dominio (DSL) para escribir pruebas potentes y mantenibles para las APIs RESTful.

Soporta cualquier método HTTP, pero tiene soporte explícito para POST, GET, PUT, DELETE, OPTIONS, PATCH y HEAD e incluye la especificación y validación, por ejemplo, de parámetros, cabeceras, cookies y body fácilmente. REST Assured es una solución de pruebas REST 100% Java que se integra perfectamente con otras herramientas como JUnit.

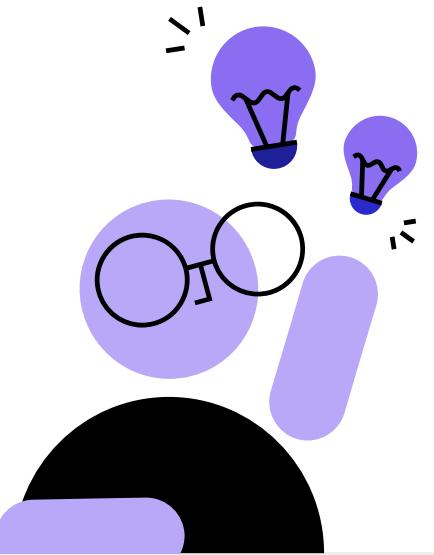
# Validando microservicios de forma automatizada con REST Assured



Para empezar a utilizar REST Assured basta con añadirlo como dependencia al proyecto. Si estamos utilizando Maven, añadimos la siguiente entrada al **pom.xml** (cambiar el número de versión para reflejar la versión que deseamos utilizar):

```
<dependency>
    <groupId>io.rest-assured</groupId>
    <artifactId>rest-assured</artifactId>
    <version>3.0.2</version>
    <scope>test</scope>
</dependency>
```

# Validando microservicios de forma automatizada con REST Assured



Después de configurar la importación de REST Assured, añadimos las siguientes importaciones estáticas a la clase de prueba:

```
import static io.restassured.RestAssured.*;  
import static org.hamcrest.Matchers.*;
```

# Acceso a API seguras

A menudo, las API están protegidas mediante algún tipo de mecanismo de autenticación. REST Assured es compatible con la autenticación básica, de compendio, de formulario y OAuth 2.0. Este es un ejemplo de cómo llamar a una API RESTful que ha sido asegurada usando autenticación básica (es decir, el consumidor de esa API necesita proporcionar una combinación válida de nombre de usuario y contraseña cada vez que llama a la API):

```
@Test
public void test_APIWithBasicAuthentication_ShouldBeGiven
Access() {

    given().
        auth().
        preemptive().
        basic("username", "password").
    when().
        get("http://path.to/basic/secured/api").
    then().
        assertThat().
        statusCode(200);
}
```

# Acceso a API seguras

Acceder a una API protegida por OAuth 2.0 es igualmente sencillo, suponiendo que se tenga un token de autenticación válido:

```
@Test
public void test_APIWithOAuth2Authentication_ShouldGive
nAccess() {

    given().
        auth().
        oauth2(YOUR_AUTHENTICATION_TOKEN_Goes_HERE).

    when().
        get("http://path.to/oauth2/secured/api").

    then().
        assertThat().
        statusCode(200);
}
```

04

# Integración continua

La integración continua (IC) es una **buenas prácticas de Agile y DevOps** en la que los desarrolladores integran sus cambios de código de forma temprana y frecuente en la rama principal o en el repositorio de código.

El objetivo es **reducir el riesgo de ver el "infierno de la integración"** esperando el final de un proyecto o un sprint para fusionar el trabajo de todos los desarrolladores. Al automatizar el despliegue, ayuda a los equipos a cumplir los requisitos empresariales, mejorar la calidad del código y aumentar la seguridad.



# Tipos de pruebas

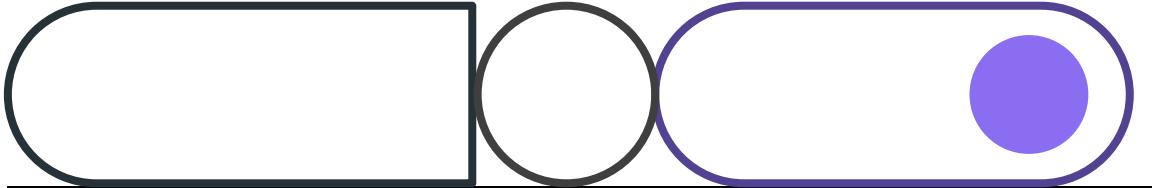
Para obtener todos los beneficios de la IC necesitaremos automatizar las pruebas para poder ejecutarlas por cada cambio realizado en el repositorio principal. Insistimos en realizar pruebas en todas las ramas de su repositorio y no solo en la rama principal. De este modo, podremos detectar los problemas con antelación y minimizar las molestias al equipo.

Hay muchos tipos de pruebas implementadas, pero no es necesario hacer todo a la vez si se está empezando. Se puede empezar con pruebas unitarias y trabajar para ampliar la cobertura con el tiempo, por ejemplo:

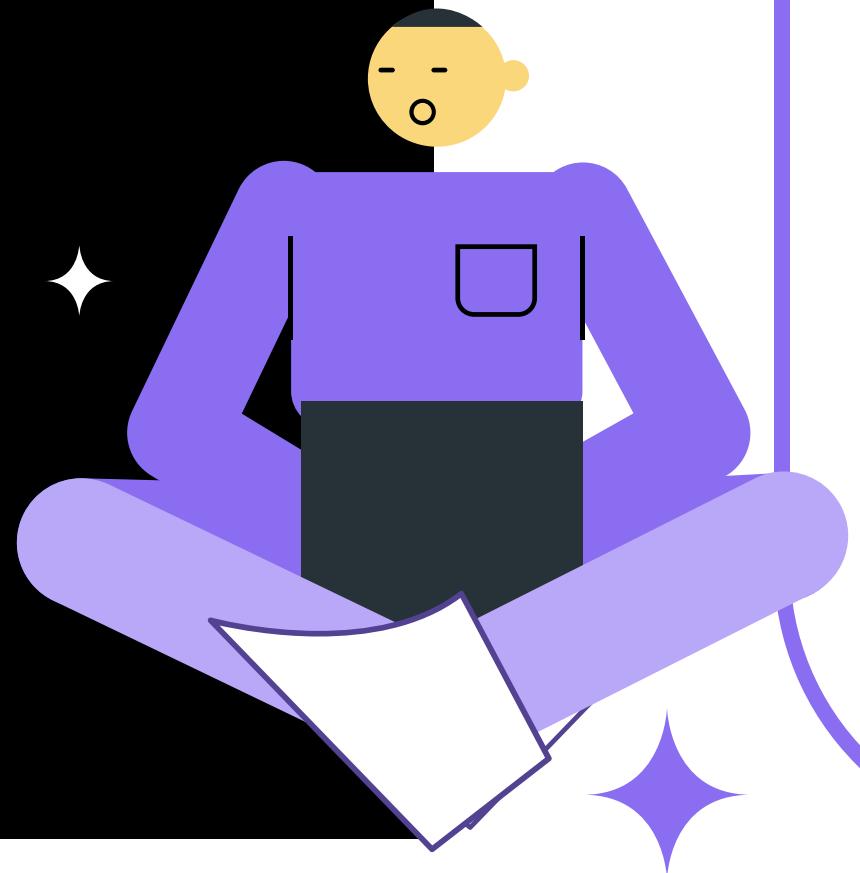
- Las pruebas unitarias tienen un alcance reducido y suelen comprobar el comportamiento de métodos o funciones individuales.
- Las pruebas de integración garantizan que varios componentes se comporten correctamente juntos. Esto puede implicar varias clases, así como probar la integración con otros servicios.
- Las pruebas de aceptación son similares a las de integración, pero se centran en los casos de negocio y no en los propios componentes.
- Las pruebas de la interfaz de usuario garantizarán que la aplicación funcione correctamente desde la perspectiva del usuario.

05

# Jenkins, pipelines y jobs



**Jenkins** es una popular herramienta de orquestación de IC que ofrece varios complementos para la integración con diversas herramientas y marcos de automatización de pruebas en el proceso de pruebas. Cuando se trata de la automatización de pruebas, Jenkins proporciona plug-ins que ayudan a ejecutar suites de pruebas, recoger los resultados de los tableros y proporcionar detalles sobre los fallos.



# ¿Por qué utilizar Jenkins para la automatización de pruebas?

- **Ejecutar conjuntos de pruebas automatizadas:** Jenkins proporciona plug-ins para varios marcos de pruebas como Selenium, Cucumber, Appium, Robot Framework, etc. Pueden integrarse en los pipelines CI para ejecutar pruebas automatizadas en cada compilación.
- **Resumir los resultados:** resume los resultados de la prueba y los muestra como una página HTML.
- **Proporciona tendencias:** Jenkins hace un seguimiento de los resultados y los muestra en forma de gráfico de tendencias. Esto proporciona una mejor visión de los resultados de las pruebas en el pasado.
- **Muestra los detalles de los fallos de las pruebas:** los resultados de las pruebas se tabulan y los fallos se registran con los resultados de las pruebas.



06

# Master test plan y release test plan

La garantía de calidad (QA) es un **proceso sistemático que asegura la excelencia de los productos y servicios**. Un sólido equipo de control de calidad examina los requisitos para diseñar, desarrollar y fabricar productos fiables, aumentando la confianza de los clientes, la credibilidad de la empresa y la capacidad de prosperar en un entorno competitivo.



# Implementación de procesos de calidad

Antes de poner en marcha un proceso de control de calidad, es importante comprender los pasos que componen una cadena de control de calidad completa e inclusiva. Aunque hay muchos pasos para implantar un sistema de garantía de calidad, los siguientes siete pasos son esenciales:

**1- Analizar los requisitos.** Cuesta más arreglar un error detectado durante las pruebas que prevenirlo en la fase de diseño de los requisitos. Los profesionales del control de calidad deben participar en el análisis y la definición de los requisitos del software, tanto funcionales como no funcionales. Las QAs deben recibir requisitos coherentes, exhaustivos, trazables y claramente marcados. Esto ayuda al equipo de control de calidad a diseñar pruebas específicamente adaptadas al software que se está probando.

**2- Planificar las pruebas.** La información obtenida durante la fase de análisis de requisitos se utiliza como base para planificar las pruebas necesarias. El plan de pruebas debe incluir la estrategia de pruebas de software, el alcance de la prueba, el presupuesto del proyecto y los plazos establecidos. También debe describir los tipos y niveles de pruebas necesarios, los métodos y herramientas para el seguimiento de los errores, y asignar recursos y responsabilidades a cada uno de los testers.

# Implementación de procesos de calidad

**3- Diseñar las pruebas.** En esta fase, los equipos de control de calidad deben crear casos de prueba y listas de comprobación que cubran los requisitos del software. Cada caso de prueba debe contener condiciones, datos y los pasos necesarios para validar cada funcionalidad. Cada prueba debe definir también el resultado esperado de la prueba para que los testers sepan con qué comparar los resultados reales.

Se recomienda que los QAs comiencen con una medida de pruebas exploratoria para familiarizarse con el software. Esto ayudaría a diseñar los casos de prueba adecuados. Si se ha definido una estrategia de automatización en el ámbito de la prueba, esta es la etapa para crear escenarios de control de calidad de la prueba de automatización.

Esta es también la etapa para preparar el entorno de prueba para la ejecución. Este entorno debe reflejar fielmente el entorno de producción con respecto a las especificaciones de hardware, software y configuraciones de red. Otras características, como las bases de datos o las configuraciones del sistema, también deben ser imitadas.

# Implementación de procesos de calidad

**4- Ejecución de pruebas y notificación de defectos.** Las pruebas comienzan a nivel de unidad con los desarrolladores que realizan pruebas de unidad. A continuación, el equipo de control de calidad realiza pruebas a nivel de la API y de la interfaz de usuario. Las pruebas manuales se realizan de acuerdo con los casos de prueba prediseñados. Todos los errores detectados se presentan en un sistema de seguimiento de defectos. Además, los ingenieros de automatización de pruebas pueden utilizar un marco de pruebas automatizado para ejecutar scripts de prueba y generar informes.

**5- Realice pruebas de repetición y de regresión.** Una vez encontrados, enviados y corregidos los errores, los QAs vuelven a probar las funciones para asegurarse de que no han pasado por alto ninguna anomalía. También realizan pruebas de regresión para verificar que las correcciones no han afectado a las funciones existentes.

# Implementación de procesos de calidad

**6- Realizar pruebas de lanzamiento.** Después de que los desarrolladores emitan una notificación de lanzamiento en la que se detalla una lista de características ya implementadas, errores corregidos, problemas recurrentes y limitaciones, el equipo de control de calidad debe identificar la funcionalidad afectada por estos cambios. A continuación, el equipo debe diseñar conjuntos de pruebas modificados que cubran el alcance de la nueva construcción.

El equipo de control de calidad también debe realizar pruebas de humo para asegurarse de que cada compilación es estable. Si la prueba se supera, se ejecutarán los conjuntos de pruebas modificados y se generará un informe al final.

**7- Aplicar mejoras continuas.** La garantía de calidad es sinónimo de mejora continua. Los resultados o la información obtenida de la encuesta de una organización o de otras herramientas de retroalimentación del cliente deben utilizarse ahora para realizar los cambios necesarios en el proceso de garantía de calidad. Esto puede implicar un mayor desarrollo del liderazgo, la formación en el servicio al cliente, una mayor dotación de personal, correcciones en el proceso de producción, cambios en el producto o servicio que fabrica o presta, etc.

# Implementación de procesos de calidad

**8- Medir resultados.** Aunque puede haber muchas razones para implantar un proceso de garantía de calidad, uno de sus principales objetivos es garantizar que la organización satisfaga las necesidades de sus clientes. Cuando una organización no lo consigue, es difícil obtener un retorno positivo de la inversión y la existencia de la organización queda en entredicho.

Desde el primer momento, hay que asegurarse de que haya objetivos cuantificables y de que todos los implicados sepan lo que hay que conseguir. Si no se alcanzan los objetivos, hay que asegurarse de que todos tengan claro qué medidas correctoras son necesarias para garantizar la seguridad y la satisfacción del cliente.

# Implementación de procesos de calidad

**9- Implementar DevOps.** DevOps aplica prácticas ágiles para los equipos de QA y Ops, simplificando la construcción, validación, despliegue y desarrollo del software. Elimina los conflictos entre los equipos de desarrollo y de control de calidad, y tiene otras ventajas:

- Proporciona un mayor control sobre el entorno de producción para los desarrolladores.
- Mejora la frecuencia de despliegue.
- Reduce la tasa de fallos de las nuevas versiones de software.
- Mejora el tiempo medio de recuperación.
- Aumenta la velocidad y la calidad del software liberado.
- Consigue un tiempo de comercialización más rápido.

# ¿Cómo redactar un buen plan maestro de pruebas?

Recordemos los seis pasos para escribir un buen plan de pruebas:

- Análisis de productos/software.
- Desarrollar la estrategia de pruebas.
- Definir los objetivos de las pruebas.
- Planificación de recursos.
- Calendario y estimación.
- Determinar los resultados de las pruebas.

07

# Métricas y reportes

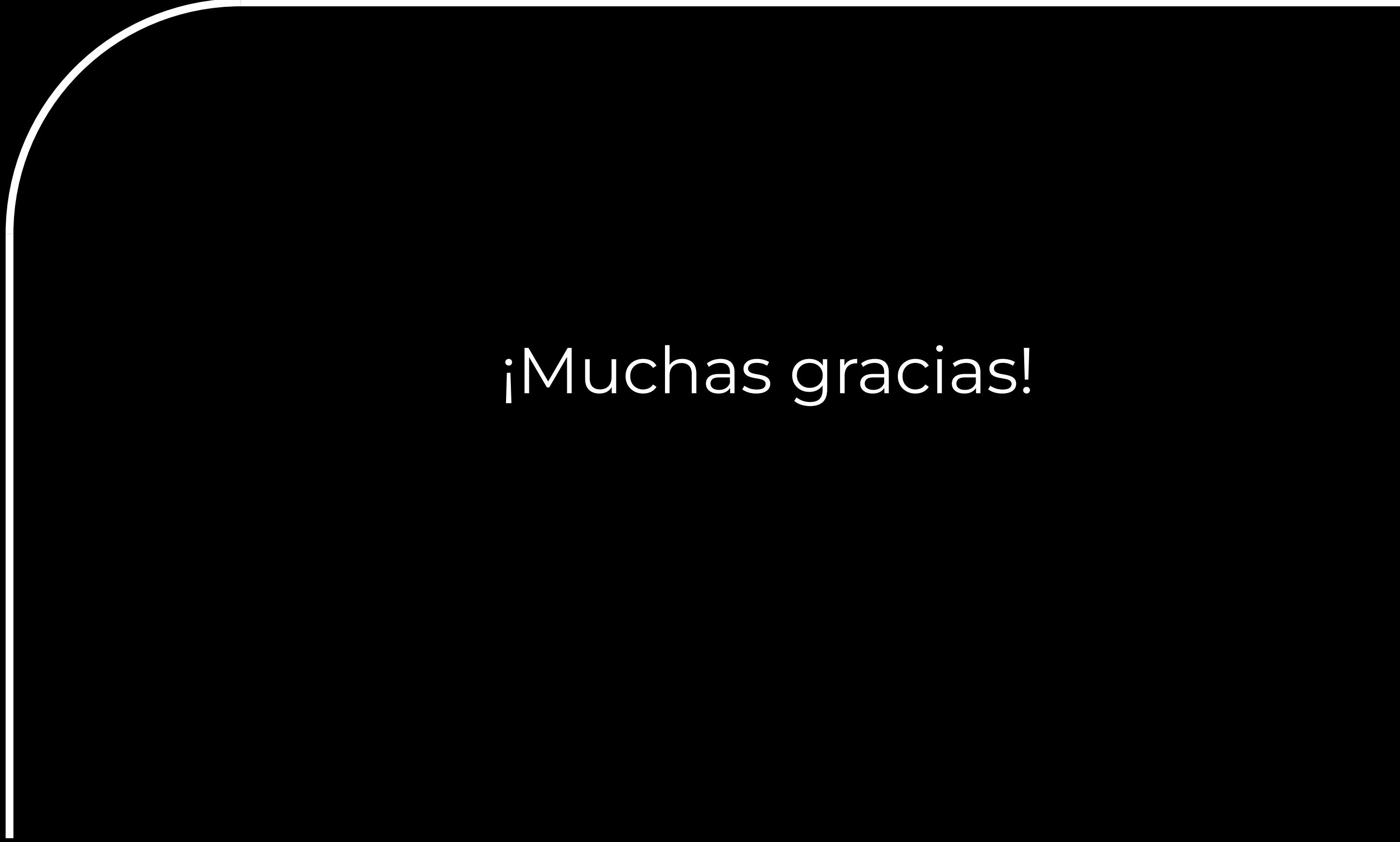
# Métricas

Naturalmente, el proceso de QA debe planificarse y supervisarse meticulosamente para que tenga éxito. La forma más eficaz de hacer un seguimiento de la eficacia de las actividades de control de calidad es utilizar las métricas adecuadas. Establecer los indicadores de éxito en la fase de planificación y hacerlos coincidir con el aspecto de cada métrica después del proceso real.

- **Métricas de prueba de control de calidad.** Normalmente, las métricas absolutas no son suficientes para cuantificar el éxito del proceso de garantía de calidad. Por ejemplo, el número de horas de prueba determinado y el número de horas de prueba reales no revelan la cantidad de trabajo que se realiza cada día. Esto deja un vacío en cuanto a la medición del esfuerzo diario realizado por los probadores al servicio de un objetivo específico de control de calidad.

# Métricas de control de calidad

- Esfuerzo de prueba.
- Cobertura de prueba.
- Economía de las pruebas.
- Equipo de pruebas.
- Distribución de defectos.



¡Muchas gracias!