Informe de Testing y Pruebas de Código
Objetivo

UNIT 1-PROGRAMMING & CODING

Hecho por: Illari Rubí Lamberti Martín
illarirubilm@msmk.university

Contenido

Introducción

El testing de software y las pruebas de código son esenciales para el ciclo de vida del desarrollo de software ya que su función es encontrar fallos en el software con tal de garantizar calidad y un buen funcionamiento del software.[1][2]

Conceptos básicos

Definición y Diferencia entre Testing y Pruebas de Código

El testing de software es un conjunto de pruebas y metodologías con las cuales se verifica y valida cada etapa de desarrollo del software, asegurándonos un correcto funcionamiento del mismo, libre de fallos y deficiencias de acuerdo a los requisitos especificados.[1]

Las pruebas de código presentan un a función parecida, ya que consisten en la realización de pruebas sobre el código, para obtener información acerca de la calidad del mismo y detectar errores.[2]

Objetivos y beneficios de realizar pruebas

El objetivo del testing de software y las pruebas de código es encontrar fallos, errores y comprobar que cumple con los requisitos del usuario

Los beneficios de hacer pruebas ayudan a la prevención de errores, la reducción de los costos de desarrollo y la mejora del rendimiento.[3]

Una buena aplicación del testing evitará una finalización del software con mala calidad, y nos permitirá desarrollar programas funcionales a la altura de los estándares de calidad deseados.[1]

Tipos de pruebas

Pruebas unitarias

Las pruebas unitarias son pruebas de bajo nivel que consisten en probar de forma individual las funcionalidad de cada una de las unidades de código.[1]

Debido su naturaleza especifica, estas son automatizadas en su mayoría por un servidor de integración continua.[5]

Cuando se planea y escriben las pruebas unitarias, es necesario haber aislado la función hasta que ya no pueda ser más dividida, por eso se llama prueba unitaria, ya que hace referencia a una unidad de código, que es el componente de prueba más pequeño del código.[3]

Este tipo de pruebas nos facilitan la documentación de las funciones del código, la optimización del mismo, ya que al ser más rápidos de ejecutar son aplicables en cada refactorización, y la integración con otras piezas de software.[1][2]

Herramientas populares[10]:

> Junit, Cactus, EasyMock, Mockito, MockEjb, Spring Test, Jetty, Dumbster.

Pruebas de integración

Las pruebas de integración son pruebas que verifican que los módulos o servicios utilizados en el software funcionan bien en conjunto.[5]

Estas complementan las pruebas unitarias y nos ayudan a validar la interacción de las piezas de un sistema entre sí.[2]

Las pruebas unitarias evalúan la cooperación, comunicación y ejecución de los elementos para identificar posibles problemas y asegurar el funcionamiento fluido del sistema.

Estas son más costosas de ejecutar que las pruebas unitarias ya que requieren que más partes del software se configuren y funcionen.[4]

Herramientas populares[11]:

> Selenium, SoapUI, Postman, Watir.

Pruebas de sistema

Las pruebas de sistemas siempre se realizan en el sistema completo, estas comprueban si el sistema cumple sus requisitos una vez integrados sus módulos y componentes individuales.[6]

Son parte de la categoría de pruebas de caja negra, lo que significa que lo único que comprueban son las características externas del funcionamiento del software, es decir su funcionamiento de cara al usuario, no el diseño interno de la aplicación.[6]

Las pruebas de sistema se realizan después de las pruebas de integración, pero antes que las de aceptación, es necesario haberlas periódicamente para garantizar un correcto funcionamiento del software durante el desarrollo del mismo.[6]

Herramientas populares[6]:

Selenium, Appium, Loadium.

Pruebas de aceptación

Las pruebas de aceptación son un conjunto de pruebas manuales, realizados después de las pruebas unitarias o de integración, destinados a verificar si un sistema cumple con los requisitos deseados.[4][5]

Estas pruebas demandan la ejecución completa de la aplicación y se centran en imitar el comportamiento de los usuarios, para rechazar cambios en caso de no cumplirse los objetivos

Además, pueden extenderse para evaluar el rendimiento del sistema.[4][5]

Es necesario definir los criterios de aceptación antes de empezar a elaborar el software y añadir cualquier cambio a estos para una buena aplicación de estas pruebas.[4]

Herramientas populares[12]:

Selenium WebDriver, Cucumber, JUnit / Serenity.

Pruebas de carga

El objetivo de las pruebas de carga es evaluar el rendimiento bajo cargas normales o esperadas, imitando escenarios realistas para comprobar un correcto funcionamiento del sistema con un tráfico normal.

Al realizar pruebas de carga, se puede evaluar el rendimiento del sistema y la eficiencia en el uso de recursos, e identificar posibles cuellos de botella.[6][7][8]

Las pruebas de carga son cruciales para garantizar que un software pueda manejar el volumen esperado de carga, proporcionando así una experiencia de usuario optimizada y evitando posibles problemas de rendimiento.

Herramientas populares[13]:

Kinsta APM, WebLOAD, Apache JMeter, LoadNinja, Loadero, smartmeter.io.

Pruebas de estrés

Las pruebas de estrés son un conjunto de pruebas realizadas durante la etapa de testing cuyo objetivo es evaluar el comportamiento y rendimiento del sistema bajo condiciones extremas de cargas elevadas en situaciones no realistas, como una carga máxima y constante durante un periodo prolongado.[7][9]

En este tipo de pruebas el resultado optimo es que el software pueda mantenerse estable y operativo bajo cargas elevadas.[7]

Sí es sistema no está optimizado, este responderá con errores o comportamientos irregulares como fallos o bloqueos de información.[9]

Son cruciales para garantizar un buen funcionamiento del sistema en momentos críticos y eventos de alta demanda.[7]

Herramientas populares[13]:

StormForge , LoadView, NeoLoad, LoadUI Pro, Silk Performer, AppLoader.

Pruebas de humo

Las pruebas de humo son pruebas básicas diseñadas para verificar el funcionamiento básico de una aplicación. Se ejecutan de manera rápida con el objetivo de proporcionar la certeza de que las funciones principales del sistema operan como se espera.[5]

Las pruebas de humo pueden ser beneficiosas al ser aplicadas justo después de completar una nueva compilación para determinar la viabilidad de realizar pruebas más exhaustivas. También se aplican inmediatamente después de una implementación para asegurarse de un correcto funcionamiento de la aplicación en el entorno recién implementado.[5]

Herramientas populares[14]:

Edición ZAPTEST ENTERPRISE, SoapUI , Testim, Robot T-Plan, Rainforest QA.

Técnicas de Testing

TDD (Test Driven Developmemt)

TDD, o Desarrollo Dirigido por Pruebas (Test-Driven Development), es una metodología de programación que implica iniciar con la escritura de pruebas, generalmente unitarias, antes de desarrollar el código fuente. Luego, se crea el código para superar exitosamente las pruebas y, finalmente, se realiza una refactorización del código escrito.[15]

Para que el Test-Driven Development sea efectivo, es necesario que el sistema sea lo bastante flexible como para permitir pruebas automáticas. Cada prueba debe ser lo suficientemente específica para determinar de manera unívoca si el código sometido a prueba cumple con los requisitos establecidos. Esta metodología favorece un diseño más eficiente al evitar el exceso de diseño en las aplicaciones, resultando en interfaces más claras y un código optimizado.[16]

Ciclo de un desarrollo de TDD [16]:

1. *Elegir un requisito:*

   Se selecciona un requisito de una lista, priorizando aquel que se considere proporcionará un mayor entendimiento del problema y, al mismo tiempo, sea de fácil implementación.

2. *Escribir una prueba:*

   Se inicia redactando una prueba específica para el requisito en cuestión. En este proceso, el programador debe comprender de manera clara las especificaciones y los requisitos de la funcionalidad que se va a implementar. Este paso obliga al programador a adoptar la perspectiva de un cliente al evaluar el código a través de las interfaces correspondientes.

3. *Verificar que la prueba falla:*

   Si la prueba no falla es porque el requisito ya ha sido implementado, o la prueba en si es errónea

4. *Escribir la implementación:*

   Para hacer que la prueba funcione se debe escribir el código más sencillo, es decir, se usa la expresión KISS ("Keep It Simple Stupid")

5. *Ejecutar las pruebas automatizadas:*

   Verificar si todo el conjunto de pruebas funciona correctamente

6. *Eliminación de duplicación:*

   El paso final es la refactorización y optimización del código para eliminar código duplicado. Esto se logra realizando pequeños cambios cada vez y luego se ejecutan las pruebas hasta que funcionen

7. *Actualización de la lista de requisitos:*

Se procede a actualizar la lista de requisitos marcando como implementado el requisito que ha sido abordado. Además, se añaden nuevos requisitos que se hayan identificado como necesarios durante este ciclo y se incorporan requisitos de diseño según sea necesario.

Características[16]:

- Uso del principio YAGNI ("You Ain't Gonna Need It")
- Escribir el mínimo código posible
- A través de pruebas se gana confianza en el código
- Tendremos un código que funcione correctamente una vez logramos hacer pasar todas las pruebas
- Requiere que el programador haga fallar los datos de prueba con el fin de asegurarse de que los casos de prueba funcionen y puedan reconocer errores

Beneficios[19]:

Usando los procesos del TDD conseguimos ser más concretos y obtenemos un código optimizado sin necesidad de esperar al final del proceso

- Conseguimos información para solucionar posibles problemas futuros a través de los feedback de API
- La escritura evolutiva del código nos permite la modificación del mismo mientras este se está creando
- Se le puede considerar una metodología ágil ya que su flujo de tareas cumple todos los requisitos necesarios para ello

Desventajas/retos[22]:

- Los equipos pueden usar TDD de manera inconsistente
- No adecuado para bases de datos grandes y complejas

- No sirve para procesos en ejecución bajo restricciones rigurosas de tiempo real, memoria, red o rendimiento.
- No adecuado para problemas del mundo real

BDD (Behaviour Driven Development)

BDD o Desarrollo guiado por el comportamiento (Behaviour-Driven Development) es un proceso de desarrollo de software surgido a partir del Desarrollo Dirigido por Pruebas (TDD, Test-Driven Development), Aunque no se trata de una técnica de testing como tal. [17][18]

Este tiene el fin de proveer al desarrollo del software de herramientas y un proceso común compartido en el desarrollo del software, para ello combina los principios y las técnicas generales del TDD con ideas del Diseño Guiado por el Dominio (DDD, Domain-Driven Design) y el Análisis/Diseño Orientado a Objetos (ADOO). [17]

La práctica del BDD requiere asistencia en el proceso de desarrollo del software, para eso se hace uso de herramientas de software especializadas.[17]

Estas herramientas sirven para automatizar el lenguaje ublicuo, usado en el Diseño Guiado por el Dominio (DDD, Domain-Driven Design) para conectar actividades entre miembros del equipo,

Además, pueden considerarse como formas especializadas de las herramientas usadas en las TDD, aunque son usualmente desarrolladas para usarse en proyectos de BDD.[17]

Características[18]:

- Todas las definiciones de BDD son escritas en un idioma común
- Principal objetivo de las definiciones es que contengan los detalles y comportamiento deseado del software

En el BDD las pruebas unitarias se pueden escribir en un lenguaje común compartido, permitiendo una comunicación eficiente entre diferentes equipos

Beneficios[18]:

defines pruebas no comportamientos

Mejor comunicación entre los equipos involucrados

curva de aprendizaje más corta

naturaleza más amigable a todos los públicos debido a una naturaleza no técnica

Desventajas/retos[21]:

- Requerimientos complejos llevan a un código enrevesado, el cual causa un código no optimizado
- Alta dependencia en los usuarios finales, en caso de no estar disponibles es complicado continuar el desarrollo del software
- Visibilidad de extremo a extremo. Debido a que es mejor separar claramente el comportamiento de prueba en escenarios específicos, esto puede llevar a fragmentos de código debidamente probados, pero es posible que no se prueben de la misma manera en que lo hará el usuario final.

DDD (Domain Driven Design)

DDD o Desarrollo guiado por el Dominio (Domain Driven Design) es un enfoque de desarrollo de software, este representa diferentes claves, metodología y patrones usados para desarrollar software al rededor del dominio.[20]

Tiene unos principios[20]:

Colocar los requisitos de la organización en el centro de la aplicación

El dominio complejo debe ser basado en un modelo de software

Se usa con el propósito de fomentar una colaboración más efectiva entre expertos del dominio y desarrolladores, permitiendo así la creación de software con objetivos claramente definidos.

Beneficios[20]:

Comunicación eficaz entre expertos del dominio y técnicos a través del lenguaje ublicuo

Foco en el desarrollo de los subdominios (secciones especificas divididas dentro del dominio) a través de Bounded Contexts.

Software más cercano al cliente ya que esta más cercano al dominio

Mantenibilidad a largo plazo

Desventajas/retos[20]:

- El aislamiento de la lógica de negocio suele costar mucho tiempo
- Se necesita un experto de dominio
- Curva de aprendizaje alta
- Solo es recomendado para aplicaciones de dominio complejo

Automatización de pruebas

Introducción y ventajas[23]

La automatización de pruebas es la detección de fallas de codificación, cuellos de botella y otros elementos que afecten al funcionamiento correcto del software mediante el uso de herramientas que ejecuten software o actualizaciones recién desarrolladas

Estas herramientas realizan las siguientes funciones:

Ejecución e implementación de pruebas

Análisis de los resultados

Comparación de la previsión con los resultados

Generar informes de rendimiento sobre el software

Mientras que las pruebas manuales son tediosas y costosas, las pruebas automatizadas:

Cuestan menos recursos y tiempo

Pueden ayudar en la detección temprana de fallos sin el riesgo de errores humanos

Fáciles de ejecutar múltiples veces tras cada cambio o incluso hasta obtener los resultados deseados

Acelera el proceso de comercialización del software

Permite la ejecución de pruebas detalladas en áreas específicas, esto hace que se puedan solucionar los problemas más habituales antes de avanzar a la siguiente etapa.

Herramientas y Frameworks populares para la automatización de pruebas

Un framework es un conjunto de herramientas y directrices convenientes a la hora de crear, ejecutar y diseñar casos de prueba.

Estas incluyen normas esenciales para las pruebas automatizadas, estas incluyen normas de prácticas, codificación y procesos para el manejo de repositorios y datos de prueba.[24]

Beneficios de los frameworks[24]:

Establecen una estrategia definida para los distintos tipos de pruebas

Optimizan la velocidad de las pruebas

Requieren poca intervención manual

Facilitan el mantenimiento del código de pruebas

Posibilitan la reutilización de código

Frameworks populares[25]:

1. *Selenium:*

   Framework de código abierto utilizado para la automatización de pruebas de aplicaciones web. Con este framework se pueden realizar pruebas de aceptación, funcionales y de integración en distintas plataformas y navegadores. A su vez, Selenium admite lenguajes de programación como Python, Ruby, Java, C#, etc,...

2. *Appium:*

   Framework de automatización de pruebas utilizado para la realización de pruebas en aplicaciones web, hibridas y nativas.

   Appium permite la ejecución de pruebas en diferentes sistemas operativos y dispositivos, haciéndolo ideal para el testeo de aplicaciones móviles

3. *Junit:*

   Framework de pruebas unitarias utilizado por java para la realización de pruebas en partes pequeñas del código fuente. Junit es integrado con herramientas de construcción como Gradle y Marven, facilitando su uso en pipelines de CI/CD

4. *TestNG:*

   Framework de pruebas unitarias e integración para Java, utilizado para la ejecución de pruebas más complejas que las realizadas con JUnit, TestNG admite características como generación de informes de prueba detallados, paralelismo y dependencias entre pruebas

Casos de uso y ejemplos:

Caso de uso[26]

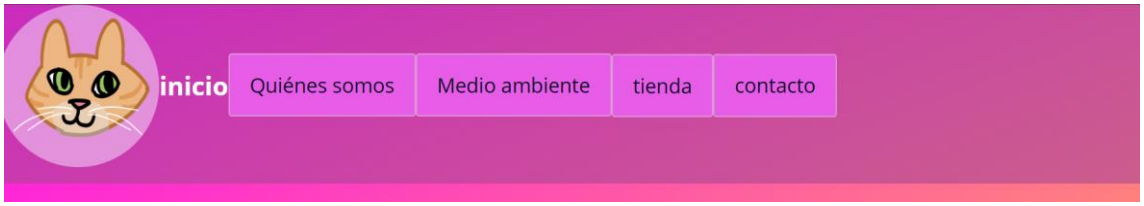
Los casos de uso cuentan la manera en que un individuo interactúa con un sistema de software con el fin de alcanzar un objetivo. Un caso de uso efectivo detalla las interacciones que conducen al logro o abandono del objetivo. En este contexto, se delinean diversas trayectorias que el usuario puede seguir durante la ejecución del caso de uso.

Varias ejecuciones del caso de uso pueden emplear los mismos o distintos escenarios.

Ejemplos:

El nav no estaba bien alineado y estaba falto de márgenes, los botones "button" no tenían espacio entre ellos

Vista:



HTML

```
<nav id="inicio" >
  <img src=" .\img\michi_logo.png" alt="logo de la tienda" class="logo">
  <h1>inicio</h1>
   <a href="html/quienes-somos.html" ><button class="mano">Quiénes somos</button></a>
   <a href="html/medio-ambiente.html"><button class="mano">Medio ambiente</button></a>
         <a  href="html/tienda.html"><button class="mano">tienda</button></a>
        <a href="html/contacto.html"><button class="mano">contacto</button></a>

    </nav>
```

CSS

```
button{
    background-color:  rgb(231, 93, 231);
    border: 2.5px solid rgb(221, 152, 215);
    border-radius: 5px;
    display:inline-block;
    padding: 0.5em;
}

nav {
    background-color: rgb(168, 72, 147);
    background-image: linear-gradient(to bottom right, rgba(255, 0, 255, 0.400), rgba(255,
255, 0, 0.400));
    padding: 20px;
    text-align: center;
    align-content: center;
    display: flex;
    align-items:center;
}
```

Intente añadir “margin” pero el código este detectaba este margin como parte de la caja, cosa que yo no deseaba por motivos de presentación frente al usuario, ya que no quiero que haya clics accidentales

Por ende, añadí una nueva clase, cree mas codigo css y modifique el HTML para que los implemente.

HTML

```
   <nav id="inicio" >
     <img src=" .\img\michi_logo.png" alt="logo de la tienda" class="logo">
     <div class="navigation">
       <h1>inicio</h1>
       <a href="html/quienes-somos.html" ><button class="mano">Quiénes somos</button></a>
       <a href="html/medio-ambiente.html"><button class="mano">Medio ambiente</button></a>
       <a  href="html/tienda.html"><button class="mano">tienda</button></a>
       <a href="html/contacto.html"><button class="mano">contacto</button></a>
     </div>
           </nav>
```

CSS

```
nav {
    background-color: rgb(168, 72, 147);
    background-image: linear-gradient(to bottom right, rgba(255, 0, 255, 0.400), rgba(255,
255, 0, 0.400));
    text-align: center;
    align-content: center;
    display: flex;
    align-items:center;
    background-attachment: fixed;
}
.navigation{
    text-align: center;
    align-content: center;
    display: inline-flex;
    align-items: center;
    margin: 1em;
    margin-left:5em;
}
.navigation :not(button) {
    margin:1em;
}
```

La funcion `.navigation :not(button)` nos permite excluir a “button” de la condición de los margin en navigation de manera que el clase “mano” no se aplica a los márgenes (mano es para que el cursor detecte como hyperlink un elemento)

Conclusión:

Las Pruebas de código y el testing de software son sumamente importantes para garantizar un buen funcionamiento del sistema a largo plazo en todos sus aspectos, en caso de usar pruebas automatizadas estas ayudan con la fluidez y productividad del desarrollo del software ya que hacen que el testeo y detección de problemas sea más rápidas que si se hacen de manera manual.

Bibliografía

1. Craft-code.com, 2021. Craft Code. [En línea]
   Available at: https://craft-code.com/que-es-el-testing-de-software/
   [Último acceso: 22 enero 2024].

2. García, G., 2021. Linkedin. [En línea]
   Available at: https://www.linkedin.com/learning/fundamentos-esenciales-de-la-programacion-2/testing-y-prueba-de-codigo?autoSkip=true&resume=false
   [Último acceso: 22 enero 2024].

3. IBM, s.f. Software testing, IBM. [En línea]
   Available at: https://www.ibm.com/es-es/topics/software-testing
   [Último acceso: 22 enero 2024].

4. Programación y mas, s.f. tipos de testing en desarrollo de software, Programación y mas. [En línea]
   Available at: https://programacionymas.com/blog/tipos-de-testing-en-desarrollo-de-software
   [Último acceso: 22 enero 2024].

5. Pittet, S., s.f. Los distintos tipos de pruebas de software, ATLASSIAN. [En línea]
   Available at: https://www.atlassian.com/es/continuous-delivery/software-testing/types-of-software-testing
   [Último acceso: 22 enero 2024].

6. ZAPTEST, s.f. *¿Qué es la comprobación de sistemas? Una inmersión en profundidad en enfoques, tipos, herramientas, consejos y trucos, ¡y mucho más!, ZAPTEST.* [En línea]
   Available at: https://www.zaptest.com/es/que-es-la-comprobacion-de-sistemas-una-inmersion-en-profundidad-en-enfoques-tipos-herramientas-consejos-y-trucos-y-mucho-mas
   [Último acceso: 23 enero 2024].

7. Ferrer, C. M., 2023. *Pruebas de carga y estrés, ¿qué son y cuándo usarlas?, paradigmadigital.* [En línea]
   Available at: https://www.paradigmadigital.com/dev/pruebas-carga-estres-que-son-cuando-usarlas/
   [Último acceso: 23 enero 2024].

8. Loadview-testing, s.f. *Pruebas de carga.* [En línea]
   Available at: https://www.loadview-testing.com/es/pruebas-de-carga/
   [Último acceso: 23 enero 2024].

9. Tamushi, 2022. *Pruebas de estrés de software: ¿qué son y para qué sirven?, TestingIT.* [En línea]
   Available at: https://www.testingit.com.mx/blog/pruebas-de-estres-de-software
   [Último acceso: 23 enero 2024].

10. Dos ideas, s.f. *Herramientas Para Pruebas Unitarias, Dos ideas.* [En línea]
Available at: https://dosideas.com/wiki/Herramientas_Para_Pruebas_Unitarias
[Último acceso: 23 enero 2024].

11. ESTEFAFDEZ, 2021. *unaqaenapuros.wordpress.com.* [En línea]
Available at: https://unaqaenapuros.wordpress.com/2021/04/28/072-pruebas-de-integracion-iii-herramientas/
[Último acceso: 23 Enero 2024].

12. TIVIT LATAM, 2022. *herramientas de automatizacion, TIVIT.* [En línea]
Available at: https://latam.tivit.com/blog/herramientas-de-automatizacion
[Último acceso: 23 enero 2024].

13. Pathak, A., 2023. *kinsta.* [En línea]
Available at: https://kinsta.com/es/blog/herramientas-pruebas-rendimiento/
[Último acceso: 23 enero 2024].

14. ZAPTEST, s.f. *¡Smoke Testing – Profundización en Tipos, Proceso, Herramientas de Software de Smoke Test y Más!, ZAPTEST.* [En línea]
Available at: https://www.zaptest.com/es/smoke-testing-profundizacion-en-tipos-proceso-herramientas-de-software-de-smoke-test-y-mas
[Último acceso: 23 enero 2024].

15. Herranz, J. I., 2011. *paradigmadigital.* [En línea]
Available at: https://www.paradigmadigital.com/dev/tdd-como-metodologia-de-diseno-de-software/
[Último acceso: 24 enero 2024].

16. Wikipedia, s.f. *Desarrollo guiado por pruebas, Wikipedia.* [En línea]
Available at: https://es.wikipedia.org/wiki/Desarrollo_guiado_por_pruebas
[Último acceso: 24 Enero 2024].

17. Wikipedia, s.f. *Desarrollo guiado por comportamiento, Wikipedia.* [En línea]
Available at: https://es.wikipedia.org/wiki/Desarrollo_guiado_por_comportamiento
[Último acceso: 24 enero 2024].

18. Vergara, S., 2019. *¿Qué es BDD (Behavior Driven Development)?, ITDO.* [En línea]
Available at: https://www.itdo.com/blog/que-es-bdd-behavior-driven-development/
[Último acceso: 24 enero 2024].

19. INESDI, 2022. *¿Qué es el TDD o Test Driven Development?, INESDI.* [En línea]
Available at: https://www.inesdi.com/blog/que-es-TDD-test-driven-

development/
[Último acceso: 24 enero 2024].

20. Loscalzo, J., 2018. *Domain Driven Design: principios, beneficios y elementos — Primera Parte, Medium.* [En línea]
Available at: https://medium.com/@jonathanloscalzo/domain-driven-design-principios-beneficios-y-elementos-primera-parte-aad90f30aa35
[Último acceso: 24 enero 2024].

21. Castro, G. C. d., 2022. *How Behaviour Driven Development improves software quality and collaboration in the development process, Medium.* [En línea]
Available at: https://medium.com/abn-amro-developer/how-behaviour-driven-development-improves-software-quality-and-collaboration-in-the-development-9fb0bb5ce1f4
[Último acceso: 24 enero 2024].

22. educative.io, 2022. *Test-driven development: What are the pros and cons?, Educative.* [En línea]
Available at: https://www.educative.io/blog/test-driven-development
[Último acceso: 24 enero 2024].

23. ZAPTEST, s.f. *¿Qué es la automatización de pruebas? Una guía sencilla y sin jerga, ZAPTEST.* [En línea]
Available at: https://www.zaptest.com/es/que-es-la-automatizacion-de-pruebas-una-guia-sencilla-y-sin-jerga
[Último acceso: 24 Enero 2024].

24. QAlified, 2022. *¿Qué es un Framework de Automatización de Pruebas? Todo lo que debes saber, QAlified.* [En línea]
Available at: https://qalified.com/es/blog/framework-automatizacion-pruebas/
[Último acceso: 24 enero 2024].

25. Gamarra, J. M. Z., 2023. *Linkedin.* [En línea]
Available at: https://www.linkedin.com/pulse/los-frameworks-m%C3%A1s-utilizados-para-el-testing-zorrilla-gamarra/?originalSubdomain=es
[Último acceso: 24 enero 2024].

26. Testeando Software, 2013. Casos de Uso vs. Casos de Prueba, Testeando Software. [En línea]
Available at: https://testeandosoftware.com/casos-de-uso-vs-casos-de-prueba/
[Último acceso: 24 enero 2024].