**FORMATO PARA EL DESARROLLO DE COMPONENTE FORMATIVO**

| PROGRAMA DE FORMACIÓN | Técnico en procesamiento de pruebas de *software* |
| --- | --- |

| COMPETENCIA | 220501099 - Probar la solución de acuerdo con parámetros técnicos y modelos de referencia. | RESULTADOS DE APRENDIZAJE | 220501099-01. Planificar las pruebas de acuerdo con el documento de requisitos. |
| --- | --- | --- | --- |

| NÚMERO DEL COMPONENTE FORMATIVO | 001 |
| --- | --- |
| NOMBRE DEL COMPONENTE FORMATIVO | Preparación de las pruebas |
| BREVE DESCRIPCIÓN | En este módulo el aprendiz comprenderá los conceptos fundamentales del plan de pruebas de *software*, cuáles son los tipos de pruebas, tipos de incidentes, errores, *bugs* y escenarios. Con esta información, el aprendiz será capaz de identificar los diferentes escenarios y componentes de *software* para establecer el plan de pruebas. |
| PALABRAS CLAVE | pruebas, incidencias, errores, *bug*, certificación |

| ÁREA OCUPACIONAL | 2 - Ciencias naturales, aplicadas y relacionadas |
| --- | --- |
| IDIOMA | Español |

1. **TABLA DE CONTENIDOS**

**Introducción**

**1. Generalidades de las pruebas de *software***

1.1. Justificación

1.2. Alcance

1.3. Objetivos de las pruebas de *software*

1.4. Ciclo de ejecución de pruebas de *software*

1.5. Casos de pruebas y escenarios

1.6. Criterios de aceptación

1.7. Niveles de aplicación de las pruebas de *software*

1.8. Tipos de pruebas

1.9. Herramientas y soportes de las pruebas de *software*

**2. Metodologías de desarrollo de proyectos**

2.1. Metodología cascada

2.2. Metodología ágil

**3. Marcos de referencia y estándares de pruebas**

3.1. Estándares en pruebas de *software*

3.2. *Test Driven Development*

3.3. Documentación de la aplicación de prueba de *software*

3.4. Criterio de manejo de lecciones aprendidas

**4. Tipos de técnicas de pruebas de *software* y herramientas para su aplicación**

**5. Conceptos de aplicación de pruebas**

5.1. *Feature*

5.2. *Issues*

5.3. *Hotfix*

5.4. *Bug*

**6. Plan de pruebas**

1. **DESARROLLO DE CONTENIDOS**

**Introducción**

Se invita a revisar el siguiente video para que comprenda algunas indicaciones iniciales del proceso de aprendizaje y los elementos de abordaje acerca de la planeación de pruebas y sus generalidades.

Imagen que contiene Interfaz de usuario gráfica

Descripción generada automáticamente

Para la elaboración de este componente, se abordaron varios autores conocidos en preparación de pruebas, de quienes se han citado y referenciado conceptos y ejemplos para los fines educativos de esta materia, en el entendido de que el conocimiento es social y, por lo tanto, es para ser usado por quienes necesitan adquirirlo. Se espera que este documento sea útil para todos aquellos, aprendices y lectores en general, que estén interesados en acercarse a asuntos básicos del procesamiento de pruebas de *software*.

**1. Generalidades de las pruebas de software**

Las pruebas de *software* se definen como una actividad en la cual un sistema, o uno de sus componentes se ejecuta en circunstancias previamente especificadas, los resultados se observan, se registran y se realiza una evaluación de algún aspecto. Hacen parte de una fase primordial en el proceso de desarrollo de *software*; en muchos casos, se definen como el método y proceso para la verificación y validación de las funcionalidades del *software*. Estas pruebas son requeridas en el control de calidad y mantenimiento de *software*, para certificar la calidad y el aseguramiento; por lo general, se acompañan de otras técnicas, como supervisiones, comprobaciones de pares o procesos automatizados, revisión de documentación de las primeras fases o formatos que no son ejecutables.

| Aplicar pruebas de *software* permitirá encontrar o identificar *bugs*, errores o incidencias, en etapas tempranas del ciclo de desarrollo, con el fin de tomar acción rápidamente, detectando niveles inadecuados de calidad o diferencias con lo establecido en su definición funcional, para así aplicar un proceso de mejora.  En el proceso de pruebas, se revisa e inspecciona el producto bajo condiciones definidas específicamente, y eventualmente se puede hacer uso de *software* especializado (CAST- Computer Aided Software Testing) como apoyo. |  |
| --- | --- |

El proceso de pruebas de *software* se debe destacar por el orden y disciplina al momento de planear y ejecutar, con indicadores, métricas y objetivos establecidos de tal manera que permitan determinar una evaluación correcta del producto; el resultado de estas actividades son los reportes que documentan la interacción entre el producto y el usuario, y hace énfasis especial en las fallas encontradas. Este proceso puede verse como una secuencia de actividades, donde cada una de sus etapas permite generar productos, artefactos, insumos asociados, recursos, e involucra distintos actores y diferentes roles.

Un primer esquema del proceso de una prueba de caja negra podría ser el siguiente:

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Con lo que conlleva el siguiente esquema de acción:

Interfaz de usuario gráfica, Aplicación, PowerPoint

Descripción generada automáticamente

Referente a los casos de prueba, es requerido que contengan las siguientes características:

**Ortogonalidad**

No tener casos que dependan de funcionalidades u otros escenarios.

**Efectividad**

Permitir la definición de elementos de entrada que permitan determinar diferentes fallos.

**Ejemplaridad**

Que “con poco se pruebe mucho”.

**Claridad**

Que permita revelar fallas de manera clara.

**1.1. Justificación**

En las pruebas de *software* es necesario contar con altos estándares de calidad, con el fin de cumplir a cabalidad las necesidades establecidas en los requisitos funcionales y los no funcionales dadas por el cliente, y poder entregar la documentación requerida.

Cualquier error en el *software* derivará en una falla que debe ser detectada a tiempo, por lo cual las pruebas deben certificar que se cuenta con el resultado esperado en la ejecución, o de lo contrario reportarlo para su corrección. Los principales objetivos que se buscan con la fase de prueba de *software* son:

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

En conclusión, es posible decir que entre más pronto se apliquen mecanismos de prueba en el proceso de desarrollo, más fácilmente podrá evitarse que el proyecto se salga del tiempo y presupuesto planeado, pues se podrán detectar los problemas originados en las fases tempranas del proceso, que son los que menor impacto tienen, se debe recordar que a mayor tiempo de desarrollo, mayor impacto al realizar ajustes o cambios.

**1.2. Alcance**

La prueba de *software* tiene limitaciones, tanto teóricas como prácticas. Desde el punto de vista teórico, la prueba se reconoce como un problema llamado no-decidible; esto significa que no se puede escribir un programa que pruebe completamente los programas sin intervención humana. Sin embargo, algunas pruebas pueden ser automatizables y apoyarse en herramientas para maximizar la búsqueda de fallas o errores importantes con los recursos asignados.

La cantidad de probabilidades o escenarios a probar en un sistema puede volverse incontrolable, por lo cual es necesario utilizar las técnicas adecuadas, pues cada método que se utilice para detectar defectos deja un residuo de defectos más sutiles contra los cuales este método es ineficaz y a esto se llama la “Paradoja del Pesticida”.

**1.3. Objetivos de las pruebas de *software***

La prueba de *software* es un recurso fundamental para la garantía del correcto funcionamiento del *software*, entre sus objetivos, se busca garantizar los siguientes aspectos:

* Detectar errores o defectos en el *software*.
* Comprobar la integración adecuada de los componentes.
* Comprobar que todos los requisitos o reglas de negocio fueron implementados correctamente, de acuerdo con lo especificado por el cliente.
* Generar un seguimiento y asegurar que los defectos encontrados se han corregido antes de liberar el *software* al cliente.
* Diseñar casos de prueba que sistemáticamente identifiquen diferentes tipos de errores o excepciones, optimizando el tiempo y el esfuerzo.

**1.4. Ciclo de ejecución de pruebas de *software***

La ejecución de pruebas de *software* está compuesta por diversas actividades, cada una con sus artefactos y ejecutores, con lo cual se busca completar el ciclo de pruebas, las cuales son:

Interfaz de usuario gráfica, Aplicación, PowerPoint

Descripción generada automáticamente

**1.5. Casos de prueba y escenarios**

Los casos de uso tienen un papel fundamental en la construcción de los casos de prueba y escenarios, dado que de ellos se derivan los artefactos y las condiciones necesarias en las cuales serán ejecutadas las pruebas (escenarios); esto permite validar, verificar y/o certificar que se han implementado a cabalidad y con calidad los requisitos funcionales que componen el producto *software*.

| Al momento de ejecutar las pruebas, se realizan validaciones de integración para funcionalidad desarrollada, con la cual se contrastan las interacciones con los demás componentes del sistema. Posteriormente, se deben realizar pruebas del sistema, con lo cual se comprueba la interacción entre los actores y el sistema como tal. | Principales certificaciones de pruebas de software en EE. UU. | ISTQB y  ASTQB? |
| --- | --- |

Se pueden identificar dos tipos de casos de prueba:

* **Basados en casos de uso:** son pruebas del sistema, de tal manera que se realizan pruebas de caja negra, comprobando con esto el comportamiento del sistema hacia el usuario de forma externa.
* **Basados en las realizaciones de los casos de uso del diseño:** corresponden a pruebas de integración, en lo cual se prueba el sistema como caja blanca, verificando cómo interactúan los componentes de forma interna.

**Escenarios:**

Los escenarios son una instancia o una representación de un caso de uso, de tal manera que permiten ejecutar un flujo completo de eventos de la funcionalidad descrita. El flujo básico de un caso de uso se define como un escenario, y este por lo general representa un *happy path*, entre otros. Cada flujo alterno se define como un escenario alterno, que puede llegar a ser un *happy path* o un *error path*. Se puede llegar a la conclusión que un caso de uso describe diferentes escenarios de uso y estos pueden definirse de la siguiente manera:

* Escenarios correspondientes al flujo básico del caso de uso o *happy path*.
* Escenarios que incluyen, por lo menos, un flujo alterno al escenario básico del caso de uso.
* Escenarios que reproducen al menos una excepción o error.

Estos escenarios determinan los lineamientos y bases para la creación de los casos de prueba.

**Generación de casos de prueba**

Un caso de prueba define qué probar y verificar en el sistema, está conformado además de un título, una descripción, que puede ser opcional, un conjunto de ítems de entrada de prueba y unos resultados esperados teniendo en cuenta las entradas proporcionadas. Se debe recordar que esto se realiza con el objetivo de probar y validar que las funcionalidades están implementadas de acuerdo con los requisitos de *software* especificados en los casos de uso. En el caso de prueba de sistema, donde se busca realizar la prueba como un todo, las metas u objetivos pueden cambiar, correspondiendo al tipo de prueba que se ejecuta, que puede ser de configuración, negativa, de instalación, entre otras.

Ivan Jacobson dio los primeros pasos basado en los casos de uso, proponiendo el siguiente flujo:

* Pruebas correspondientes al flujo básico del caso de uso o el *happy path*.
* Pruebas correspondientes a flujos alternos.
* Pruebas específicas que surgen de requerimientos específicos para una instancia del caso de uso.
* Pruebas asociadas a las características descritas en documento asociado al caso de uso.

Más adelante, en el 2001, Jim Heuman, trabajador de Rational SW Corporation, publica un artículo en el cual se describe y define el procedimiento con el cual generar casos de prueba a partir de casos de uso; dicho proceso basado en RUP, el cual consta de tres pasos:

* Determinar el conjunto de escenarios, para cada caso de uso.
* Identificar al menos un solo caso de prueba y las condiciones en las cuales será ejecutado, para cada escenario.
* Determinar los valores de entrada que serán usados, para cada caso de prueba.

**Generación de los escenarios**

Este paso se debe realizar con la mayor profundidad posible, dado que, al definir buenos escenarios de caso de uso, se estarán garantizando posteriormente buenos escenarios de prueba, con lo cual, para poder definir de manera asertiva los escenarios de prueba, debe estar totalmente clara y detallada la descripción del caso de uso, de tal manera que permita identificar la combinación del flujo básico y los flujos alternos. Hay que tener presente que estos flujos son aquellos que constituyen los escenarios.

**Identificación de los casos de prueba**

Dado que ya se tienen planteados todos los escenarios de los casos de uso, se procede a obtener los casos de prueba, con lo cual se debe analizar cada uno de los escenarios y, al mismo tiempo, referenciarse en la descripción detallada de los casos de uso, para despejar dudas o aclarar detalles. Debe generarse como mínimo un caso de prueba para cada escenario, pero la generalidad es que se definan más. Si el caso de uso no contiene la información requerida para dar detalles del comportamiento o funcionalidad específica, dado que se hayan omitido datos importantes, para lograr que las pruebas tengan la calidad necesaria para una certificación exitosa, será necesario aclarar dicha descripción y buscar completar la información que sea requerida para que sea considerada como completa.

**Identificación de los datos de entrada**

Cuando ya se tengan definidos todos los casos de prueba, estos deberán ser revisados y validados para asegurar su precisión, y así identificar los casos de prueba redundantes u aquellos que posiblemente hayan sido omitidos. Luego, se utilizarán los datos de entrada reales de la prueba y se reemplazarán en la definición del caso de prueba. Estos datos deberán corresponder a datos reales usados en el proceso de ejecución de pruebas.

**1.6. Criterios de aceptación**

Los criterios de aceptación definen los requisitos del cliente, indicando cómo debe comportarse el producto *software* para una determinada acción que se pueda llevar a cabo, generalmente por una acción desencadenada por un usuario de la aplicación, respondiendo si se ha construido correctamente el producto o si se ha construido el producto correcto.

Deben describir un contexto, una acción o evento y una consecuencia o respuesta esperada del sistema. La forma más utilizada para describir los criterios de aceptación es la conocida como *Given-When-Then*. Aquí un ejemplo:

* **Dado** un usuario que no ha iniciado sesión en el sistema.
* **Cuando** intenta ingresar a alguna funcionalidad reservada o privada.
* **Entonces** el sistema debe redireccionar automáticamente a la página de inicio de sesión para que pueda ingresar.

Aunque expresan el comportamiento del producto *software*, siempre es utilizado un lenguaje de negocio, no técnico. En el anterior ejemplo, se usó “**Cuando** intenta ingresar a alguna funcionalidad reservada o privada” y no “Cuando hace clic en alguna de las opciones del menú que están restringidas por el inicio de sesión”.

La precisión en la definición de los criterios de aceptación depende de la madurez y experiencia del equipo de desarrollo.

**Recuerde:** hacer uso de los criterios de aceptación en la definición de requisitos de *software* o historias de usuario, dependiendo de la metodología empleada, permitirá tener claro el alcance de la implementación y cuándo finaliza, la funcionalidad de lo construido es lo que el cliente requería y, además, lo que el usuario necesita.

**1.7. Niveles de aplicación de las pruebas de *software***

Los niveles de pruebas están categorizados por las fases de diseño, teniendo en cuenta la planificación de los requisitos funcionales, hasta las pruebas anteriores a la puesta en producción.

Cada fase en el proceso de desarrollo de pruebas está asociada a una de las etapas de construcción de *software*. Aunque no es estricto que ambas fases trabajen en paralelo, sí deberán estar en constante comunicación y mutua colaboración.

En la siguiente figura, se presenta el modelo V que se aplica al proceso de desarrollo de pruebas de *software*:

**Figura 1**

*Modelo en V en asignación de tipos de pruebas*

****

Nota. Adaptada deAmrani (2019)**.**

Es una representación de los diferentes tipos de pruebas. En la base de la pirámide, están las **pruebas unitarias,** que cubren la mayor parte del código de forma aislada, permitiendo la prueba por unidades o módulos independientes; su implementación se da de manera sencilla e instantánea. Mientras que en la punta de la pirámide, se ubican las **pruebas de aceptación,** que requieren de la integración de todos los elementos que conforman el programa; su implementación es más compleja y su ejecución requiere más tiempo.

**1.8. Tipos de pruebas**

Las pruebas de *software* se pueden clasificar en dos tipos principalmente:

* Pruebas funcionales.
* Pruebas no funcionales.

Teniendo en cuenta los diferentes frentes y aspectos de un producto *software*, con lo cual se requerirán diferentes tipos de pruebas, tales como: pruebas unitarias, pruebas de integración, pruebas de estrés, pruebas de rendimiento, pruebas de escalabilidad, entre otras. Cada uno de estos tipos de pruebas de *software* determina la visibilidad de la aplicación, desde el código fuente hasta la interacción y experiencia con el usuario.

Entrando en detalles, algunos de los tipos de pruebas más comunes son:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

En conclusión, es posible decir que las pruebas de rendimiento son útiles para verificar el rendimiento del producto *software*; las pruebas funcionales se aplican para verificar que las funcionalidades están de acuerdo con lo establecido en el documento de requisitos funcionales y hacen lo que en los casos de uso está especificado; las pruebas de integración sirven para verificar la sinergia e integración entre los diferentes componentes y módulos descritos en un producto *software*; y las pruebas unitarias se definen para comprobar si son correctos los elementos como las clases, funciones, módulos o componentes que son tratados como una unidad.

**1.9. Herramientas y soportes de las pruebas de *software***

| Son llamados “*testers*” aquellas personas encargadas de poner a prueba la calidad y el correcto funcionamiento de un producto *software*; esto se ejecuta por medio de una planificación previa, con la finalidad de mantenerse dentro de los tiempos establecidos y el presupuesto disponible.  Para realizar este rol, es necesario establecer los objetivos y la planeación, con el fin de simplificar, automatizar y agilizar las pruebas que se llevarán a cabo en los ciclos de desarrollo. |  |
| --- | --- |

Es posible encontrar diversidad de herramientas que dan soporte, trazabilidad y control a las pruebas de *software,* y que apoyan el ciclo de vida de la construcción del producto *software* (conocido como SDLC); estas ayudan a la administración de la documentación, automatización, soporte y los artefactos requeridos para la última etapa, conocida como pruebas de regresión. Todo esto permite un adecuado control del proceso y seguimiento de los avances, gestión de errores, *bugs* e incidencias, y la generación de informes de acuerdo con las pruebas realizadas. Las herramientas usadas en el proceso de pruebas de *software* se clasifican teniendo en cuenta la etapa en la cual se encuentra el proyecto.

En el siguiente recurso de aprendizaje, se mencionan algunas:

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

El uso de estas herramientas también puede generar riesgos, dado que se está limitado a la calidad de estas y dado que, si presentan algún error, pueden generar una validación errónea en la verificación del producto *software*. Sin embargo, la mayor ventaja es la optimización de tareas y el incremento en la calidad de las pruebas, con lo que se potencializa la certificación del producto y el control del proceso.

**2. Metodologías de desarrollo de proyectos**

Son aquellas que indican qué métodos y técnicas hay que usar en cada fase del ciclo de vida de desarrollo del proyecto; básicamente, son un marco de trabajo usado para planificar, estructurar y controlar los procesos, a medida que va avanzando el proyecto.

**2.1. Metodología cascada**

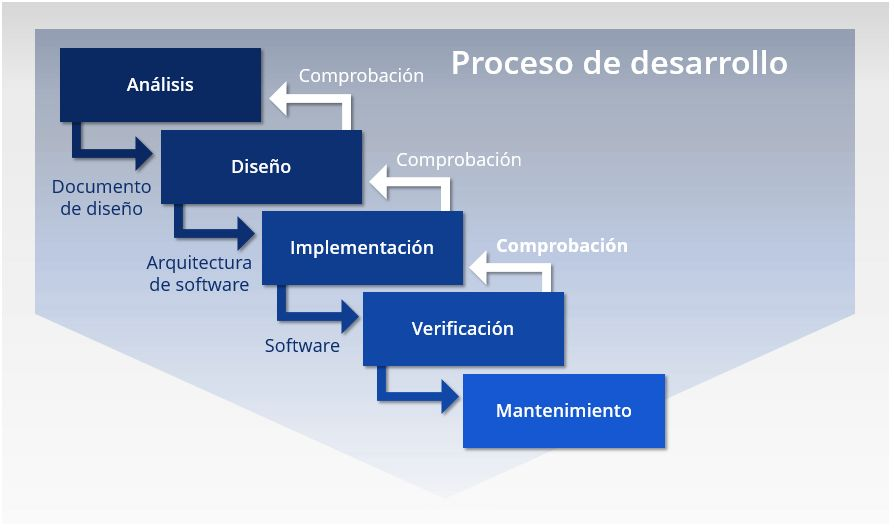
Es un proceso lineal, donde se separan los procesos del desarrollo de *software* en diferentes fases, que son sucesivas en el contexto de la ejecución del proyecto. A diferencia de los modelos iterativos, cada una de las fases se ejecuta una sola vez y el resultado de cada fase es un insumo para la siguiente.

Esta metodología es usada especialmente en el proceso de desarrollo de *software*.

El modelo es atribuido al teórico de la informática Winston W. Royce, aunque Royce no es el creador de este modelo; al contrario, en su ensayo de 1970 titulado *Managing the Development of Large Software Systems*, el teórico proporciona una reflexión constructiva acerca de los procedimientos lineales y, a modo de alternativa, presenta un modelo iterativo incremental en el que cada una de las fases se basa en el resultado de la fase anterior y valida los resultados de esta.

**Figura 2**

*Representación de implementación de metodología en cascada*



Nota. Adaptada de Romero (2018).

En esta imagen, se presenta la metodología en cascada, con cada una de sus fases o etapas, dado que no es posible acceder a la fase de diseño si no se ha terminado la fase de requisitos, por lo cual son de naturaleza bloqueante. Este tipo de metodologías también son conocidas como tradicionales, porque fueron los primeros modelos que se incorporaron en el proceso de desarrollo de *software*, aunque con el tiempo se identificaron las problemáticas, al necesitar obtener productos a corto plazo.

Amplíe la información sobre cada una de las fases de la metodología en cascada:

Interfaz de usuario gráfica, Aplicación, PowerPoint

Descripción generada automáticamente

**2.2. Metodología ágil**

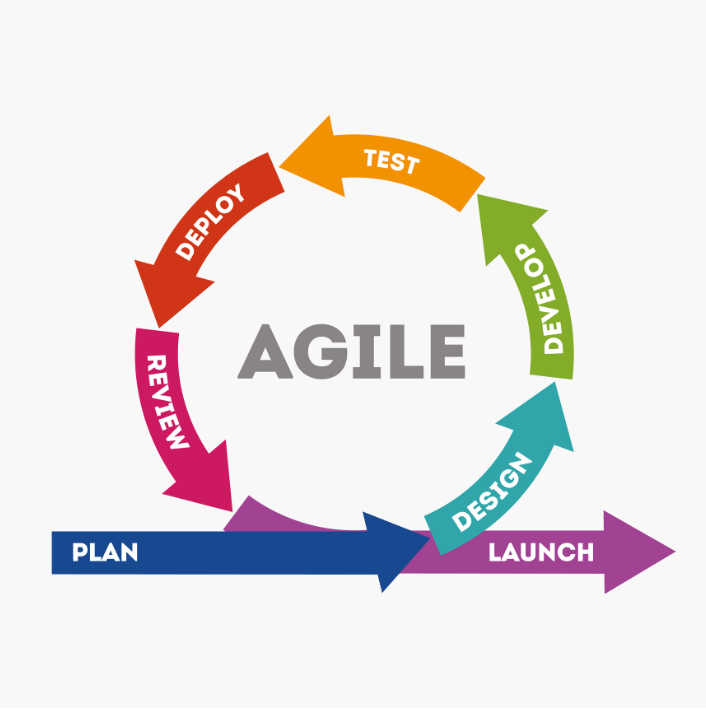
La vertiente ágil en el desarrollo de *software* pretende distribuir de forma permanente y continua el proceso de desarrollo en iteraciones rápidas. Sin embargo, el término “metodología ágil” es engañoso, dado que implica que el enfoque ágil es la única manera de abordar el desarrollo de *software*. En esta vertiente no se define una serie de pasos o de indicaciones sobre qué hacer en el proceso de desarrollo de *software*; por el contrario, se trata de una forma de pensar en colaboración los flujos de trabajo, en la cual se definen una serie de valores que orientan las decisiones respecto a lo que se hace y la forma en que se hace.

| Las metodologías ágiles buscan generar una serie de piezas que agregan valor y satisfacción al cliente; en estas se implementan enfoques flexibles y de trabajo en equipo para mejorar el desempeño del proceso, con la finalidad de ofrecer mejoras constantes. | Ver las imágenes de origen |
| --- | --- |

En la siguiente figura se puede observar el flujo de trabajo de una iteración en la metodología ágil, dado que el ciclo de vida de la construcción de un producto *software* está compuesto por muchas iteraciones que, a su vez, son pequeñas, medibles y alcanzables. Básicamente, consiste en dividir el proyecto en entregables pequeños, los cuales se diseñan, desarrollan, certifican, despliegan y entregan al cliente, lo que permite que, a medida que el producto va creciendo, los nuevos *features* que se van agregando son probados y certificados con calidad y se recibe un *feedback* del cliente de forma rápida. Este tipo de metodologías son denominadas iterativas incrementales, dado que con cada ciclo el producto va creciendo.

***Figura 3***

*Representación de una iteración en la implementación de una metodología ágil*



Nota. Adaptada de Cognodata (2019).

La metodología ágil nació aproximadamente en el año 2001, dando respuesta a los proyectos gestionados por medio de metodologías en cascada, por lo cual una serie de desarrolladores redactó el manifiesto ágil. En este manifiesto se describieron cuatro características principales, las cuales se deberían priorizar por encima de cualquier otra cosa. Es así como los equipos que trabajan bajo esta vertiente deben valorar lo siguiente:

* Las personas y las interacciones antes que los procesos y las herramientas.
* El *software* en funcionamiento antes que la documentación exhaustiva.
* La colaboración con el cliente antes que la negociación contractual.
* La respuesta ante el cambio antes que el apego a un plan.

**3. Marcos de referencia**

Estos aportan a la mejora de los procesos, entregando recomendaciones de mejores prácticas, teorías y lineamientos aplicables a un contexto determinado, para las pruebas de *software* también se cuentan con estándares que sirven como guía y referencia.

**3.1. Estándares en pruebas de *software***

El objetivo de trabajar bajo un marco de estándares es hacer uso de documentos, artefactos y métodos que han sido validados previamente por entidades y organizaciones a nivel internacional. Entre estos es posible destacar el estándar IEEE-829 de 2008, que posteriormente fue reemplazado por la ISO/IEC/IEEE 29119-3:2013, la cual será reemplazada por la ISO/IEC/IEEE FDIS 29119-3; esta última está actualmente en desarrollo.

En la actualidad podemos hacer referencia a la ISO/IEC/IEEE 29119-3:2013, en la cual se han definido una serie de ítems acordados internacionalmente para la aplicación y ejecución de pruebas de *software* para cualquier organización, incluye plantillas y admite las pruebas dinámicas, pruebas funcionales, no funcionales, pruebas manuales, pruebas automatizadas, pruebas con y sin secuencias de comandos. Los artefactos definidos en la ISO/IEC/IEEE 29119-3:2013 se pueden utilizar en cualquier momento del ciclo de vida del desarrollo de *software*.

La ISO/IEC/IEEE 29119-3:2013 está compuesta de 5 partes:

Aplicación

Descripción generada automáticamente con confianza baja

**3.2. *Test Driven Development***

Es el desarrollo dirigido por pruebas (TDD por sus siglas en inglés), es un proceso de desarrollo relativamente nuevo, que está comenzando a ganar terreno y a ser conocido en el ámbito del desarrollo de *software* en las organizaciones a nivel mundial. Se basa en evidencias anecdóticas y en una serie de evaluaciones empíricas. Hablando del desarrollo tradicional, las pruebas se usan para validar y verificar la calidad del *software*, y su creación se realiza después de que existe la funcionalidad.

En TDD, las pruebas, adicionalmente al tema de validación y verificación, se usan con fines de especificación, donde primero se desarrolla la prueba y posteriormente la funcionalidad. Esto permite que la calidad del *software* aumente y la productividad del desarrollador mejore, dado que se lleva un mejor esfuerzo en depurar y se avanza rápidamente con el progreso del producto *software*.

En el siguiente recurso de aprendizaje se presentan los beneficios del desarrollo guiado por pruebas (TDD):

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Ahora que ya conoce los beneficios que implica aplicar TDD, observe el ciclo completo del proceso, revisando cada uno de sus pasos. En la siguiente figura se puede observar el flujo de manera gráfica:

**Figura 4**

*Flujo de TDD*



Nota. Adaptada de Díaz (2013).

Amplíe la información sobre cada uno de los pasos del TDD.

Interfaz de usuario gráfica, Aplicación, PowerPoint

Descripción generada automáticamente

**3.3. Documentación de la aplicación de prueba de *software***

La documentación de productos de *software* son artefactos importantes, que permiten transferir y comunicar aspectos que, al revisar o inspeccionar un sistema, no se pueden entender rápidamente y suelen ser complicados.

Esta se puede categorizar en función de las personas, roles o grupos a los cuales está dirigida:

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

**3.4. Criterio de manejo de lecciones aprendidas**

| Las lecciones aprendidas son el conocimiento adquirido sobre una actividad, tema o proceso por el cual se vivieron diferentes experiencias, reflexiones y análisis, que permiten tener un concepto crítico sobre los aspectos que pudieron afectar positiva o negativamente la situación evaluada. |  |
| --- | --- |

Estas reflexiones, por lo general, dan una ventaja competitiva entre actores que compiten en una misma industria, además permiten capturar las evidencias para identificar tendencias, relación entre causa y efecto, están delimitadas a un contexto en particular, sugieren recomendaciones, prácticas, utilidades para la aplicación o replicación del nuevo conocimiento en diferentes contextos; se busca tener resultados similares a los experimentados previamente.

**4. Tipos de técnicas de pruebas de *software* y herramientas para su aplicación**

Estas técnicas y herramientas permiten evaluar el *software* de acuerdo con los requerimientos y necesidades de los usuarios, por esto es necesario conocer los tipos, enfoques y particularidades.

**Caja negra**

**Partición de equivalencia:** se establecen objetivos de cobertura teniendo en cuenta las entradas y las salidas. Las entradas pueden ser manuales, vía interfaces, o parámetros de interfaz en las pruebas de integración.

En esta técnica es importante identificar las clases de equivalencia, por ejemplo, la suma entre 2 y 4, es decir que cualquier valor diferente a 6 será un valor no válido. Luego se generan los casos de prueba con diferentes valores, para asegurar que la funcionalidad que se está verificando solo acepte los valores de 2 y 4 que retorna como resultado el valor de 6.

En la siguiente figura podemos observar que, en el caso de las pruebas de caja negra, no es necesario conocer qué hay dentro de la caja (código fuente del producto *software*), como se explicó previamente, solo interesa lo que ingresa y lo que sale de dicha funcionalidad, en la imagen representado con las flechas.

****

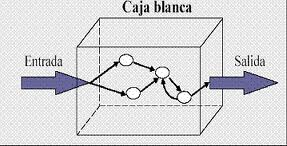
Nota. Adaptada de PMOinformatica (2016).

**Caja blanca**

Estos tipos de pruebas se basan en una estructura identificada dentro del producto *software*, según unos niveles específicos, que son:

* **Nivel de componente:** evalúa la estructura de un componente *software*, por ejemplo: sentencias, condiciones, flujos alternos.
* **Nivel de integración:** evalúa la estructura en un árbol de llamadas y se basa en el diagrama en el que se especifica cómo un módulo llama a los otros módulos.
* **Nivel de sistema:** evalúa aspectos que van de cara al usuario, por ejemplo, la estructura de un menú, si este permite conectar con el hipervínculo definido o si está estructurado de tal manera que representa un ítem del menú.
* **Basadas en el flujo de control:** se cubren todos los bloques o sentencias de código en el producto *software*, así como las combinaciones definidas en ellas, de tal manera que se puede medir en porcentaje; se obtiene una cobertura del 100% cuando todo el código fuente del producto *software* pasó por lo menos una vez por alguna de las pruebas.

En la figura, se puede observar la representación de las pruebas de caja blanca. En este caso, a diferencia de las pruebas de caja negra, sí interesa cuál es el contenido de la caja, pues este tipo de pruebas busca inspeccionar cómo está definida la funcionalidad para obtener los resultados teniendo en cuenta los datos de entrada. En esta parte es importante resaltar que se tienen en cuenta las buenas prácticas de desarrollo, optimización y reutilización de código, adecuado uso de nomenclaturas, entre otros.

****

Nota. Adaptada de Garay (2013).

**Pruebas de rendimiento**

Permiten determinar el comportamiento de un sistema en términos de estabilidad y respuesta con respecto a un escenario en particular, referente a la carga de trabajo, concurrencia, ancho de banda, características particulares de equipos o servidores, como RAM, CPU. También permiten identificar, medir, investigar, validar o verificar otros aspectos de calidad, tales como la seguridad, escalabilidad y el consumo de recursos. Estas pruebas constituyen unos estándares de ejecución en la implementación, diseño y arquitectura de un producto software, algunas herramientas de pruebas de rendimiento que puede tomar como referencia son: Jmeter y Gatling.

**Pruebas de seguridad**

Son la serie de actividades que son ejecutadas para encontrar fallas y vulnerabilidades, con la finalidad de minimizar el impacto de ataques, acceso a información susceptible y la pérdida de información.

Estas pruebas son primordiales dado que los productos *software* deben asegurar siempre la confidencialidad, disponibilidad e integridad de los datos y funcionalidades que maneja el producto. Es importante denotar lo crítico que sería la pérdida o corrupción de esta información en el contexto empresarial; por este motivo, estas pruebas deben ir de la mano y ejecución en todas las fases del proyecto.

**Pruebas de fiabilidad**

Cada producto *software*, dependiendo del contexto de negocio, deberá estar disponible 24/7; dar soporte a la sobrecarga de concurrencia, por ejemplo, en un cyberday; que una entidad financiera esté disponible en las fechas de pagos, como quincenas o fines de mes; entre otros ejemplos.

En estos escenarios, las pruebas de fiabilidad permiten verificar la tolerancia a fallos de sistema, disponibilidad, escalabilidad. Todas estas pruebas se realizan para verificar los requerimientos de infraestructura, que son los que soportan el producto *software*.

Las pruebas se aplican principalmente a los componentes de infraestructura, tales como redes, servidores y *software* de soporte, al producto *software*, a las integraciones con segundos o terceros, y no menos importante, a los procesos masivos y críticos del sistema.

**Pruebas unitarias**

Se aplican en los elementos más pequeños de un producto *software* y son considerados como una unidad, sin tener en cuenta la interacción entre otros componentes; cada componente es probado para verificar su correcto funcionamiento, donde se desarrolla una única función cohesiva. El objetivo de la prueba unitaria es analizar y verificar cada pequeña parte y certificar que funciona correctamente.

Este tipo de pruebas son usadas comúnmente por desarrolladores que están en la vertiente ágil, especialmente programadores extremos, los partidarios de TDD. Actualmente, se han vuelto muy populares y requeridas por las empresas, por lo cual más desarrolladores se están incorporando a la práctica de crear pruebas unitarias en sus desarrollos.

Las herramientas para realizar pruebas unitarias son: Junit, Phpunit, Mocha, Mockito, Chai, Sinon, Jasmine, Jest, Ava y TestNG, Powermock, XCTest

**Pruebas de integración**

Son una extensión de las pruebas unitarias. Dos o más unidades que ya han sido probadas, conjugadas en un componente y sus interfaces interactúan entre ellas. El objetivo de estas pruebas es certificar la funcionalidad y la seguridad de la comunicación entre las unidades, componentes y paquetes que componen el producto *software*. Esto es beneficioso, dado que permite determinar la eficiencia de las pruebas en trabajo conjunto, pues no importa cómo está definida cada prueba unitaria si los componentes entre sí no están integrados, y conlleva una falla del sistema.

Las herramientas para realizar pruebas de integración son: Mocha, Jasmine, Jest y Ava

**Pruebas de estrés**

Estas pruebas se implementan para validar el comportamiento del *software* en situaciones críticas y poco usuales, por ejemplo, una saturación de memoria, el llenado del almacenamiento provisto para el sistema, restricción en el acceso a los recursos, caídas inesperadas de la conexión a Internet, sobrecarga de funciones o peticiones de procesamiento.

**5. Conceptos de aplicación de pruebas**

Es importante reconocer los diferentes conceptos que se usan para la aplicación de prueba de *software*, con el fin de que pueda familiarizarse con estos términos técnicos.

**5.1. *Feature***

FOSD es un paradigma en el desarrollo de *software* orientado a *Features*, éste sirve para la construcción de sistemas basados en características propias.

| Una *feature* (característica) es una unidad del producto *software* que da respuesta a un requisito funcional, la cual puede generar una opción de implementación y representa una decisión de diseño. |  |
| --- | --- |

Se utiliza este tipo de modelos para definir funciones comunes y las variantes de una serie de categorías de productos de *software*. Los modelos de *features* son usados para describir las propiedades, funcionalidades, alternativas u opcionales dentro de un dominio.

El objetivo de trabajar bajo este paradigma es organizar el producto *software* en módulos de *feature*, los cuales representan características o funcionalidades comunes que discrepan de otras f*eatures*, y a este conjunto de *features* se le denomina **Línea de Productos de *Software*** (SPL). Por lo cual, una SPL es una familia de funcionalidades relacionadas con un dominio en particular, en la que los artefactos de implementación son compartidos.

Para la implementación de una SPL, es necesario analizar el dominio y las diferencias que existen entre otras *features*.

**Programación Orientada a *Features* (FOP)**

Es una forma de trabajo en la programación que permite la representación de las características similares y las variaciones de un producto de *software*, donde el concepto principal está basado en *features*, y en la que, partiendo de un grupo de características, se pueden generar muchos sistemas de software que compartan características comunes y difieran en otras.

Este tipo de programación nace a partir de las limitaciones que la programación orientada a objetos posee en relación con la implementación de una SPL.

**5.2. *Issues***

En el ámbito del desarrollo de *software*, un *issue* se constituye en una unidad de trabajo. Puede ser una tarea que busca:

* Solucionar un fallo.
* Una característica solicitada.
* La solicitud de una documentación específica.
* Otro tipo de solicitud al equipo de desarrollo.

Una de las características principales de los *issues* es que manejan un ID único, que se utiliza principalmente para la trazabilidad, es decir, para poder hacer seguimiento del estado y poder relacionar con otros *issues* del proyecto.

A continuación, conozca los diferentes tipos de relaciones que es posible encontrar entre los *issues*.

Interfaz de usuario gráfica, Aplicación, PowerPoint

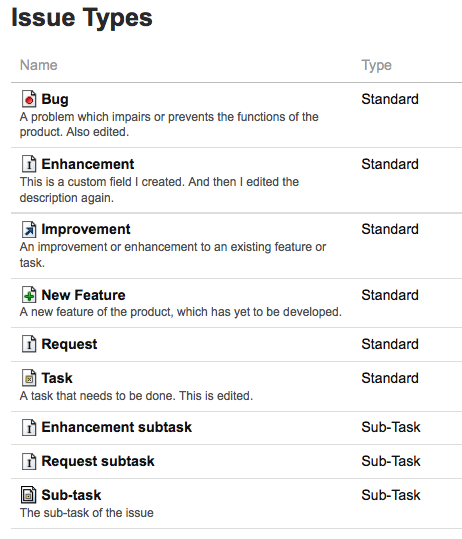
Descripción generada automáticamente

Como se especificó anteriormente, un *issue,* en realidad, es una generalización de tareas que no solo se relaciona con defectos, sino que también hace referencia a cualquier tipo de actividad o elemento en el escenario de desarrollo.

En la siguiente imagen, se presentan algunos ejemplos, los cuales se pueden adecuar a las necesidades del proyecto u organización.

**Figura 5**

*Ejemplo de tipos de issues*



Ahora, los *issues* se pueden categorizar de la siguiente forma:

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Cada categoría maneja su propio flujo de trabajo, dado que los tipos no son iguales y posiblemente no manejen los mismos estados.

**5.3. *Hotfix***

| El *hotfix* hace referencia a una solución rápida, conocida como un parche rápido o un parche en caliente. Estos, por lo regular, se aplican en productos *software* que están operando en producción y que atienden a reportes de fallas o defectos de cara al usuario final, por lo cual no se aplican en los equipos de desarrollo. Para el desarrollador implica que el cambio se debe generar rápidamente y fuera del proceso y ciclo normal de la implementación de *software*. Esto puede aumentar el costo al tener que requerir un desarrollo ágil, que posiblemente se deba ejecutar en horas extra y tomando medidas urgentes. |  |
| --- | --- |

**Tenga en cuenta:** lanzar un *hotfix* es una solución arriesgada y la menos indicada para solucionar un problema, dado que esto puede generar una pérdida inmediata del servicio y, dependiendo de la urgencia y la complejidad, se deberá tomar la decisión de crear un *issue* para resolver el problema y posponer el *hotfix*.

**5.4. *Bug***

| Los *bugs* representan una falla o un error en el producto *software*, se tratan de una inconsistencia en la ejecución de alguna funcionalidad y desencadenan con ello un resultado inesperado o un resultado no deseado. En ese caso, se realiza el apoyo de programas especializados para encontrar estas fallas y con ello brindar la solución pertinente; a esas herramientas se les denomina depuradores. |  |
| --- | --- |

Un *bug* informático se puede establecer desde la programación o el diseño de *software*, e implica que en algún momento se manifestará hacia el usuario.

Algunos ejemplos de estas fallas pueden ser, entre otros:

* Objetos o variables que no fueron inicializadas.
* No indexación de las bases de datos.
* Un ciclo que no tiene delimitador y se convierte en un ciclo infinito.

La regla es que en el momento de ejecución y compilación del producto *software*, los entornos de desarrollo organicen e identifiquen en primera instancia los *bugs,* para que se realice su correctivo antes de pasar a algún ambiente.

**6. Plan de pruebas**

El plan de pruebas de *software* se establece con la finalidad de atender los objetivos de calidad en el marco del desarrollo de *software*, definiendo los aspectos relacionados con los módulos o funcionalidades a verificación, tipos de prueba, ambientes, recursos asignados, entre otros aspectos necesarios en el proceso de prueba. Presenta el enfoque para realizar la verificación de los componentes del producto. Como se ha mencionado previamente, todo proceso de desarrollo de *software* es importante que cuente con la implementación de la fase de pruebas.

El personal que se dedica a ejecutar pruebas de *software* requiere de un plan de pruebas de *software*, el cual tiene como objetivo:

**Comunicar a todos los involucrados del proyecto los entregables, ítems a ser certificados, criterios de aprobación y fallos, criterios de suspensión y reanudación, las necesidades de ambiente, las capacitaciones necesarias para los integrantes del equipo, riesgos y el laboratorio de usabilidad.**

El plan se puede aplicar a todo el proyecto y se ajusta a las necesidades de cada empresa, teniendo en cuenta el tamaño del proyecto, el tiempo, el costo, el ciclo de vida del *software* y los involucrados.

Cada entidad puede definir su propio plan de pruebas basado en buenas prácticas:

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

**Formatos**

Recordemos que el estándar ISO/IEC/IEEE 29119-3:2013 nos provee de una serie de artefactos validados a nivel internacional por las organizaciones a la vanguardia del proceso de desarrollo de *software* y, a su vez, al margen de la ejecución de pruebas de software.

**Tabla 1**

*Formatos*

Documentación de proceso de pruebas (a escala) organizacional

* Política de pruebas
* Estrategia organizacional de pruebas

Documentación del proceso de gestión de pruebas

* Plan de pruebas (incluyendo una estrategia de pruebas)
* Reporte de estado de prueba
* Reporte de compleción de pruebas

Documentación del proceso de pruebas dinámicas

* Especificación de diseño de prueba
* Especificación de caso de prueba
* Especificación de procedimiento de prueba
* Requerimientos de datos de prueba
* Reporte de preparación de datos de prueba
* Requerimientos de entorno de prueba
* Reporte de preparación de entorno de prueba
* Resultados reales - Resultados de prueba
* Registro de ejecución de prueba
* Reporte de incidentes de prueba

1. **ACTIVIDADES DIDÁCTICAS (OPCIONALES SI SON SUGERIDAS)**

| DESCRIPCIÓN DE ACTIVIDAD DIDÁCTICA | |
| --- | --- |
| Nombre de la Actividad | Identificación de los tipos de pruebas de *software* |
| Objetivo de la actividad | Evaluar la comprensión de los conceptos, permitiendo que el aprendiz pueda reconocer los diferentes tipos de pruebas de *software* y cómo se aplicaron. |
| Tipo de actividad sugerida |  |
| Archivo de la actividad  (Anexo donde se describe la actividad propuesta) | Anexo\_CF01\_ActividadDidactica |

1. **MATERIAL COMPLEMENTARIO:**

| Tema | Referencia APA del Material | Tipo de material  (Video, capítulo de libro, artículo, otro) | Enlace del Recurso o  Archivo del documento o material |
| --- | --- | --- | --- |
| Generalidades de las pruebas de *software* | Bolaños, D., Sierra, A. y Alarcón, M. (2007). *Pruebas de software y JUnit. Un análisis en profundidad y ejemplos prácticos*. Pearson Prentice Hall. <http://www.ebooks7-24.com.bdigital.sena.edu.co/?il=4716&pg=21> | Libro - Capítulo 1: Fundamentos de las pruebas de *software* (p. 21 - 32). | <http://www.ebooks7-24.com.bdigital.sena.edu.co/?il=4716&pg=21> |
| Metodologías de desarrollo de proyectos | Contreras, M., Herrera, M. y Gallardo, O. *Capítulo III Pruebas Automáticas para Evaluar Cursos de Programación de Computadores*. Tecnológico de Antioquia. <https://dspace.tdea.edu.co/bitstream/handle/tdea/1251/INGENIERIA%20VOLUMEN%203%20FINAL.pdf?sequence=1&isAllowed=y#page=30> | Libro - Capítulo I análisis de las metodologías ágiles para el desarrollo de software | <https://dspace.tdea.edu.co/bitstream/handle/tdea/1251/INGENIERIA%20VOLUMEN%203%20FINAL.pdf?sequence=1&isAllowed=y#page=30> |
| Tipos de técnicas de pruebas de *software* y herramientas para su aplicación | Pérez, Y., Gallegos, J., Zapata, S., Ccama, D. y Choque, R. (2020). Design Thinking en la Planificación de Pruebas de Software. Innovación y Software. *Revista Innovación y Software, 1*(2), p. 40-51. <https://revistas.ulasalle.edu.pe/innosoft/article/view/24/13> | Artículo | <https://revistas.ulasalle.edu.pe/innosoft/article/view/24/13> |
| Tipos de técnicas de pruebas de *software* y herramientas para su aplicación | Uribe, C. (2020). *Desarrollo de aplicaciones web y pruebas de software.* Tecnológico de Antioquia. <https://dspace.tdea.edu.co/bitstream/handle/tdea/1081/INFORME%20FINAL.pdf?sequence=1&isAllowed=y> | Documento | <https://dspace.tdea.edu.co/bitstream/handle/tdea/1081/INFORME%20FINAL.pdf?sequence=1&isAllowed=y> |

1. **GLOSARIO**

| TÉRMINO | SIGNIFICADO |
| --- | --- |
| Automatización: | proceso en el cual se utilizan herramientas tecnológicas casi sin la necesidad de la intervención de personas. |
| CAST: | significa “prueba de *software* asistida por computadora”, es la herramienta que se usa para aplicar las pruebas. |
| Caso de uso: | es una herramienta en la cual se define un flujo con una serie de secuencias que dan paso a un resultado de valor observable. |
| Escenario: | lugar con una serie de condiciones y variables, en el cual se desarrolla un suceso o acción. |
| *Framework*: | es un marco de trabajo, representa un conjunto de conceptos, funciones, prácticas y criterios que están regulados y estandarizados. Generalmente, orientados a resolver temas en particular y que sirven para resolver problemáticas con características similares. |
| *Happy path*: | es un anglicismo y es usado para expresar el escenario ideal de un producto *software* o una funcionalidad, representa un escenario donde todo ocurre según las especificaciones técnicas y nada sale mal, logrando de esta manera el objetivo esperado. |
| Módulo: | es una parte de un producto *software*, de las muchas cosas que debe realizar un programa para cumplir con funciones o tareas. |
| *Mock*: | son recursos y objetos que están preprogramados para que cumplan con una expectativa en específico, dado esto, siempre retornan un valor estático para una invocación puntual. |
| *Plugin*: | es un componente o aplicación que presenta una integración o sinergia con otra, generalmente se usan para agregar una nueva función. |
| Requisito: | atributo, cualidad, circunstancia, cosa o comportamiento que es necesario para algo. |
| SUT: | el Sistema Bajo Prueba (SUT) es el producto *software* que representa todos los actores, en sí, clases, paquetes, módulos. |

1. **REFERENCIAS BIBLIOGRÁFICAS**

Alarcón, C. (2019). *Pruebas de software*. EcuRed. <https://www.ecured.cu/index.php?title=Pruebas_de_software&oldid=3536813>.

Amrani, D. (2019). *Niveles de pruebas de software*. La región vulnerable. <https://medium.com/la-region-vulnerable/niveles-de-pruebas-de-software-ac9a888311f7>

Cognodata. (2019). *12 principios de la metodología agile en el desarrollo de proyectos*. <https://www.cognodata.com/blog/principios-metodologia-agile-desarrollo-proyectos>

Díaz, D. (2013). *TDD, BDD & Test de aceptación*. Adictos al trabajo. <https://www.adictosaltrabajo.com/2013/05/27/tdd-bdd-test-de-aceptacion/>

Garay, G. (2013). *Téc D P - Técnicas basadas en la estructura o técnicas de caja blanca (K4)*. Scrum-QA. <http://scrum-qa.blogspot.com/2013/04/tecnicas-de-diseno-de-pruebas-tecnicas_12.html>

PMOinformatica. (2016). *Pruebas de caja negra ISTQB*. <http://www.pmoinformatica.com/2016/04/pruebas-caja-negra-istqb.html>

Romero, E. (2018). *Actividades del desarrollo de software en el modelo de cascada*. ResearchGate. <https://www.researchgate.net/figure/Figura-1-Actividades-del-desarrollo-de-software-en-el-modelo-de-cascada_fig1_320935254>

1. **CONTROL DEL DOCUMENTO**

|  | Nombre | Cargo | Dependencia | Fecha |
| --- | --- | --- | --- | --- |
| Autor(es) | Gustavo Rodriguez | Experto temático | SENA Colombo Alemán. | Septiembre 2021 |
| Giovanna Andrea Escobar Ospina | Diseñadora Instruccional | Regional Antioquia - Centro de Servicios de Salud | Septiembre 2021 |
| Ana Catalina Córdoba Sus | Revisora Metodológica y Pedagógica | Regional Distrito Capital – Centro para la Industria de la Comunicación Gráfica. | Octubre 2021 |
| Rafael Neftalí Lizcano Reyes | Asesor pedagógico | Regional Santander - Centro Industrial del Diseño y la Manufactura. | Octubre 2021 |
| Darío González | Corrección de estilo | Regional Tolima – Centro Agropecuario La Granja | Octubre 2021 |

1. **CONTROL DE CAMBIOS**

**(Diligenciar únicamente si realiza ajustes a la Unidad Temática)**

|  | Nombre | Cargo | Dependencia | Fecha | Razón del cambio |
| --- | --- | --- | --- | --- | --- |
| Autor (es) |  |  |  |  |  |