UNIVERSIDAD NACIONAL DE ITAPÚA

FACULTAD DE INGENIERÍA

Ingeniería en Informática

Trabajo Final de Grado

Desarrollo de un Sistema para la ejecución distribuida de pruebas unitarias de aplicaciones Ruby on Rails utilizando la librería Rspec

Rodrigo Jacinto Fernández Ojeda - Leslie Fabiana López Figueredo

Encarnación - Paraguay

2022

UNIVERSIDAD NACIONAL DE ITAPÚA

FACULTAD DE INGENIERÍA

Ingeniería en Informática

Desarrollo de un Sistema para la ejecución distribuida de pruebas unitarias de aplicaciones Ruby on Rails utilizando la librería Rspec

Rodrigo Jacinto Fernández Ojeda - Leslie Fabiana López Figueredo

Trabajo Final de Grado presentado a la Facultad de Ingeniería de la Universidad Nacional de Itapúa, cuyo tema fue aprobado por Res. Dec. 049/2019.

Tutor: Prof. Ing: Aldo Miguel Medina Venialgo

Encarnación - Paraguay

2022

UNIVERSIDAD NACIONAL DE ITAPÚA

FACULTAD DE INGENIERÍA

Ingeniería en Informática

HOJA DE EVALUACIÓN DE TFG

INTEGRANTES DE LA MESA EXAMINADORA

* \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
* \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
* \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
* \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
* \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

CALIFICACIÓN FINAL: \_\_\_ (\_\_\_\_\_\_\_)

ACTA Nº:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

FECHA: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Secretaria General Decano

Encarnación - Paraguay

2022

Dedico este trabajo a mi familia y a mi esposo por estar siempre a mi lado, por la confianza y el apoyo incondicional que me brindaron, sin ellos no hubiera sido posible cumplir esta meta.

Leslie López

Dedico este trabajo a mis padres Silvino y Juana por haberme apoyado durante toda la carrera, y vaya que me tomé mi tiempo, y a mi esposa Maga por estar más emocionada que yo por la culminación de este trabajo .

Rodrigo Fernández

AGRADECIMIENTOS

Índice

[1 INTRODUCCIÓN 13](#__RefHeading___Toc378_132173762)

[1.1 PROBLEMÁTICA 13](#__RefHeading___Toc380_132173762)

[1.2 OBJETIVOS 17](#__RefHeading___Toc382_132173762)

[1.2.1 General 17](#__RefHeading___Toc384_132173762)

[1.2.2 Específicos 17](#__RefHeading___Toc384_1321737621)

[1.3 Alcance del Proyecto 18](#__RefHeading___Toc380_1321737623)

[1.4 Justificación 20](#__RefHeading___Toc380_13217376231)

[2 MARCO TEÓRICO 22](#__RefHeading___Toc386_132173762)

[2.1 CICLO DE VIDA DEL DESARROLLO DEL SOFTWARE 22](#__RefHeading___Toc388_132173762)

[2.1.1 Modelos De Proceso De Software Prescriptivo O Tradicionales 22](#__RefHeading___Toc390_132173762)

[2.2 Pruebas 26](#__RefHeading___Toc388_1321737621)

[2.2.1 Pruebas De Software 27](#__RefHeading___Toc390_1321737621)

[2.2.2 Niveles De Pruebas 28](#__RefHeading___Toc390_13217376211)

[2.2.3 Métodos De Testeo 29](#__RefHeading___Toc29729_1259762573)

[2.3 Desarrollo dirigido 30](#__RefHeading___Toc3366_3631554615)

[2.3.1 Desarrollo Dirigido Por Pruebas De Aceptación (ATDD) 30](#__RefHeading___Toc3212_21706708331)

[2.3.2 Desarrollo Dirigido Por Comportamiento (BDD) 30](#__RefHeading___Toc3214_21706708331)

[2.3.3 Desarrollo Dirigido Por Pruebas (TDD) 32](#__RefHeading___Toc3216_21706708331)

[2.4 Pruebas Unitarias 33](#__RefHeading___Toc388_13217376211)

[2.4.1 Librerías Para Testeo Unitario En Diferentes Lenguajes 33](#__RefHeading___Toc390_132173762111)

[2.4.2 Frameworks De Pruebas Unitarias En Ruby 34](#__RefHeading___Toc3981_2392111844)

[2.4.3 Comparativa Entre Rspec Y Minitest 34](#__RefHeading___Toc25151_872665549)

[2.5 Protocolos 35](#__RefHeading___Toc3220_2170670833)

[2.5.1 Función De La Capa De Transporte 35](#__RefHeading___Toc3222_2170670833)

[2.5.2 Protocolo De Control De Transmisión (TCP) 36](#__RefHeading___Toc3224_2170670833)

[2.5.3 Protocolo De Datagramas De Usuario (UDP) 38](#__RefHeading___Toc3226_2170670833)

[2.5.4 Protocolo De Copia Segura (SCP) 39](#__RefHeading___Toc3228_2170670833)

[2.6 SOCkets 39](#__RefHeading___Toc3230_2170670833)

[2.6.1 Sockets Según Su Orientación 40](#__RefHeading___Toc3232_2170670833)

[2.6.2 Arquitectura Cliente - Servidor 40](#__RefHeading___Toc3234_2170670833)

[2.7 DIFUSIÓN DE DATOS 41](#__RefHeading___Toc3236_2170670833)

[2.7.1 Unidifusión (Unicast) 41](#__RefHeading___Toc3238_2170670833)

[2.7.2 Multidifusión (Multicast) 42](#__RefHeading___Toc3240_2170670833)

[2.7.3 Difusión (Broadcast) 42](#__RefHeading___Toc2849_1483933243)

[3 Conclusión 44](#__RefHeading___Toc386_1321737621)

Tabla de Figuras

[Figura 1: Prototipo de la arquitectura del sistema implementado. 19](#Figura!0|sequence)

[Figura 2: Modelo en Cascada. 24](#Figura!1|sequence)

[Figura 3: Modelo Incremental. 25](#Figura!2|sequence)

[Figura 4: Modelo V. 26](#Figura!3|sequence)

[Figura 5: Ranking de frameworks de pruebas unitarias en Ruby. 31](#Figura!4|sequence)

[Figura 6: Estructura de BDD 34](#Figura!5|sequence)

[Figura 7: Cabecera de un segmento TCP. 37](#Figura!6|sequence)

[Figura 8: Cabecera de un datagrama UDP. 38](#Figura!7|sequence)

[Figura 9: Comunicación entre sockets. 40](#Figura!8|sequence)

[Figura 10: Ejemplo básico de Cliente-Servidor. 41](#Figura!9|sequence)

[Figura 11: Envío de paquete, utilizando la técnica Unicast. 42](#Figura!10|sequence)

[Figura 12: Envío de paquete utilizando Broadcast. 43](#Figura!11|sequence)

Lista de Tablas

Tabla 1: Librerías de testeo unitario existentes para ciertos lenguajes. 14

Tabla 2: Comparativa de herramientas que soportan la ejecución distribuida de pruebas unitarias. 15

Lista de Abreviaturas

ATDD Desarrollo Orientado a Pruebas de Aceptación (Acceptance Test Driven Development)

BDD Desarrollo Guiado por el Comportamiento (Behavior Driven Development)

PU Pruebas unitarias

PC Computadora personal (Personal computer)

TCP Protocolo de Control de Transmisión (Transmission Control Protocol)

TDD Desarrollo Guiado por Pruebas (Test Driven Development)

UDP Protocolo de datagramas de usuario (User Datagram Protocol)

SCP Protocolo de copia segura (Secure Copy Protocol)

TFG Trabajo Final de Grado

Lista de S**ímbolos**

Resumen

En la actualidad las pruebas unitarias son una herramienta fundamental para el aseguramiento de la calidad de nuestro software. A medida que se avanza en la elaboración de un software, la cantidad y complejidad de las pruebas unitarias aumentan y las cualidades de la computadora del desarrollador son casi siempre las mismas que al inicio del proyecto, esto incide negativamente en el tiempo de ejecución de las mismas.

En el presente trabajo final de grado se desarrolló un sistema que permitirá al programador ejecutar pruebas unitarias de código para aplicaciones Ruby on Rails utilizando la librería RSpec en una arquitectura distribuida, es decir, la carga de trabajo que conlleva la ejecución de estas pruebas será distribuida en una red de computadoras con el fin de descentralizar la carga de trabajo y aumentar la velocidad de ejecución y obtención de resultados de las pruebas unitarias. Para implementar el Sistema se utilizaron distintas herramientas como sockets TCP y UDP, para la detección y conexión automática entre las distintas máquinas (agentes) y la máquina principal (máster), el protocolo SCP para el envío de archivos, y la gema Rspec para la ejecución de pruebas unitarias.

**Palabras claves:** Pruebas unitarias, Rspec, arquitectura distribuida, agentes, máster.

Abstract

1. INTRODUCCIÓN
   1. PROBLEMÁTICA

Actualmente el software está presente en múltiples objetos cotidianos como son los teléfonos celulares, computadoras, medios de transporte, electrodomésticos, etc.

Un modelo típico de proceso de desarrollo de software tiene generalmente varias etapas: planificación y recopilación de información, análisis y definición de requisitos, diseño y definición de la arquitectura, desarrollo, pruebas según los requisitos y la perspectiva técnica, despliegue y mantenimiento (Chowdhury, Bhowmik, Hasan y Rahim, 2017)⁠.

Durante el proceso de desarrollo de software, continuamente se realizan pruebas con el objetivo de verificar el correcto funcionamiento del software. Según Gómez, Jústiz y Delgado (2013)⁠ las pruebas contribuyen a la calidad del software y estas pruebas deben comenzar desde el momento en el que el desarrollador inicia la implementación del software y deben finalizar antes del despliegue del mismo.

Los principales tipos de pruebas que se pueden realizar a cualquier tipo de software son: pruebas unitarias, prueba de integración, pruebas de regresión, pruebas de humo y pruebas del sistema (Pauta Ayabaca y Moscoso Bernal, 2017)⁠.

La prueba unitaria enfoca los esfuerzos de verificación en la unidad más pequeña del diseño de software: el componente o módulo de software (Pressman, 2010)⁠. Las pruebas de integración comprueban que los componentes individuales funcionen correctamente luego de integrarlos. Las pruebas de regresión comprueban que los cambios efectuados en una parte no afecten a otras partes del software. Las pruebas de humo comprueban el software constantemente con el objetivo de reducir los riesgos de errores al momento de la integración. Las pruebas del sistema están enfocadas directamente en los requisitos de negocio, verificando el ingreso, procesamiento, recuperación de los datos y la implementación propiamente dicha (Pauta Ayabaca y Moscoso Bernal, 2017).

Las pruebas unitarias normalmente son el primer nivel de prueba de un sistema de software y están motivadas por el hecho de que el costo de encontrar y corregir errores en el momento de la prueba de la unidad, es menor que encontrar y corregir errores que se encuentran durante las pruebas de integración, las pruebas del sistema o después del despliegue, las pruebas unitarias facilitan las pruebas de regresión cuando el software cambia porque permiten a los desarrolladores verificar que no hayan dañado la funcionalidad existente (Ganesan et al., 2013)⁠. La popularidad de las pruebas unitarias ha ido aumentando a medida que fueron surgiendo más librerías de testeo, para los distintos lenguajes de programación existentes (Athanasiou, Nugroho, Visser y Zaidman, 2014)⁠. En la 1 se presentan algunas librerías que permiten implementar pruebas unitarias para ciertos lenguajes de programación, a estas librerías también se le denominan frameworks.

|  |  |
| --- | --- |
|  | Librerías de Testeo |
| Java | Junit5 , TestNG |
| .Net | NUnit |
| Ruby | UnitTest, Rspec, RubyUnit, MiniTest |
| Phyton | Unitest, PyUnit, doctest |
| PHP | SimpleTest, PHPUnit |
| Javascript | Qunit, Mocha, Jasmine, Chai |

Tabla 1: *Librerías de testeo unitario existentes para ciertos lenguajes.*

Elaboración propia.

En nuestra experiencia, cuando el desarrollador requiere ejecutar todas las pruebas unitarias de manera local, el tiempo que lleva la ejecución de estas pruebas unitarias está limitado por las características de procesamiento de su computadora. Es por ello que a medida que va creciendo el software, aumentan la cantidad de pruebas unitarias, por ende, el tiempo de ejecución de las pruebas unitarias también aumentan.

En la actualidad, es común utilizar herramientas de integración continua durante el proceso de desarrollo de software. La integración continua consiste en la fusión del trabajo de los desarrolladores, permitiendo mejorar la calidad del software (Fitzgerald y Stol, 2017, citado por Shahin, Ali Babar y Zhu, 2017)⁠⁠. Esta práctica incluye la ejecución automatizada de las pruebas de software (Leppänen et al., 2015, citado por Shahin et al., 2017)⁠.

En la 2 se presenta una comparativa de algunas herramientas de integración continua que mejoran la velocidad de ejecución de pruebas unitarias. Esta mejora se logra distribuyendo las pruebas a través de varias computadoras interconectadas, de esta manera comparten la carga de trabajo y reducen el tiempo de ejecución. Estas herramientas dependen de un repositorio para la obtención del código fuente sobre el cual ejecutar las pruebas, por lo que no soportan la ejecución de pruebas sobre el código del desarrollador, es decir, el código sobre el cual está trabajando el desarrollador en su computadora local.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Jenkins | TeamCity | Travis CI | Circle CI | Codeship | Dist Test |
| Intefaz | Web | Web | Web | Web | Web | Web |
| Lenguajes Soportados | Varios | Varios | Varios | Varios | Varios | Java |
| Instalación | Linux, Windows, OS X | Linux, Windows, OS X | Servicio en la nube | Servicio en la nube y servidor privado en Linux o OS X | Servicio en la nube | Linux |
| Licencia | MIT, Código Abierto | Comercial, Gratis bajo ciertas condiciones | Comercial, Gratis bajo ciertas condiciones | Comercial | Comercial | Apache License 2.0 |

Tabla 2: *Comparativa de herramientas que soportan la ejecución distribuida de pruebas unitarias.*

Elaboración propia.

Según Wang (2016)⁠, cuando la ejecución de las pruebas unitarias toma demasiado tiempo se generan los siguientes problemas:

* La productividad del desarrollador se ve afectada porque tiene que esperar la ejecución de todas las pruebas antes de continuar su trabajo.
* El desarrollador se vuelve reacio a agregar más pruebas porque incrementa el tiempo de espera para la ejecución de todas las pruebas.
* La calidad del software baja porque se dejan de escribir pruebas.
* El desarrollador pierde tiempo actualizando las pruebas tratando de disminuir su tiempo de ejecución.
* El desarrollador termina considerando a las pruebas más como una carga que una ventaja adicional para mejorar la calidad.

En 2016, Wang desarrolló un framework denominado Dist Test, orientado al desarrollador, que permite la ejecución distribuida de pruebas unitarias para software desarrollado en el lenguaje Java (Wang, 2016)⁠. Este tipo de herramientas enfocadas al desarrollador pueden llegar a ser bastante útiles, pero dar soporte a la gran variedad de lenguajes de programación existentes es prácticamente imposible, por esta razón se propone el desarrollo de un Sistema similar al mencionado por Wang que dé soporte a software desarrollado en lenguaje Ruby, enfocándose específicamente en la librería de testeo RSpec para Ruby on Rails.

* 1. OBJETIVOS
     1. General

Desarrollar un Sistema que permita reducir el tiempo de ejecución de pruebas unitarias de aplicaciones Ruby on Rails utilizando la librería RSpec.

* + 1. Espec**íficos**
* Analizar y definir los requerimientos del Sistema.
* Implementar el algoritmo de distribución de pruebas unitarias.
* Implementar el algoritmo de distribución de código fuente.
* Configurar el entorno de prueba para el Sistema.
* Implementar las pruebas unitarias que se usarán para evaluar el Sistema.
* Evaluar los resultados obtenidos de la ejecución del Sistema.
  1. Alcance del Proyecto

Se desarrollará un Sistema, que permitirá al desarrollador ejecutar pruebas unitarias de código para aplicaciones Ruby on Rails utilizando la librería RSpec. La ejecución de pruebas unitarias se realizará utilizando una arquitectura distribuida, es decir, la carga de trabajo que conlleva la ejecución de estas pruebas será distribuida en una red de computadoras.

El Sistema podrá ser utilizado desde una línea de comandos. Una vez iniciado el proceso de ejecución de pruebas unitarias sobre el código del desarrollador, se irán mostrando los resultados en la interfaz de línea de comandos, indicando las pruebas ejecutadas con éxito o que hayan fallado.

El Sistema podrá obtener el código fuente sobre el cual se ejecutarán las pruebas unitarias desde dos ubicaciones posibles, una de ellas es la computadora del desarrollador y la otra será desde un repositorio Git.

El sistema soportará la ejecución de pruebas unitarias que requieren conexión a base de datos. Al iniciar un proceso de ejecución de pruebas se creará una base de datos de prueba y al finalizar el proceso la base de datos será eliminada, cada proceso trabajará sobre una base de datos diferente.

El Sistema estará compuesto por dos aplicaciones, la primera aplicación a la que se denominará Aplicación Coordinadora, se encargará de coordinar el proceso de ejecución de las pruebas, y mostrar los resultados, la segunda aplicación a la que se denominará Aplicación Asistente, será un servicio en segundo plano que se encargará de la ejecución de las pruebas bajo las órdenes de la Aplicación Coordinadora. Para que un desarrollador pueda iniciar la ejecución de sus pruebas, deberá tener instalado la Aplicación Coordinadora. En la 1 se presenta una arquitectura compuesta por cuatro computadoras, la Aplicación Coordinadora instalada en PC1 coordinará la ejecución de las pruebas, comunicándose con las Aplicaciones Asistentes, las cuales estarán ejecutándose como un servicio dentro de todas las computadoras de la red, eventualmente, las computadoras que componen la red pueden tener instalada una o ambas aplicaciones.

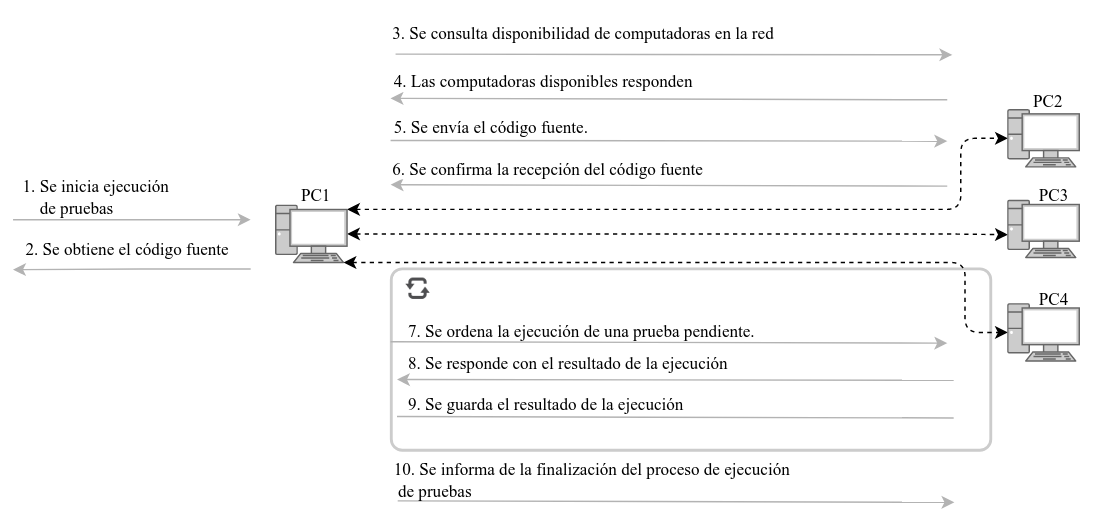


Figura 1: *Prototipo de la arquitectura del sistema implementado.*

Elaboración propia.

* 1. Justificaci**ón**

En grandes proyectos, la ejecución de todas las pruebas después de cada cambio puede convertirse en una operación costosa que requiere varias horas. (Blondeau et al., 2017)⁠.

Según Blondeau et al. (2017)⁠, en una importante empresa de TI, se encontraron proyectos con un entorno tan complejo y con tantas pruebas que se necesitaban horas para ejecutarlas todas, por este motivo los desarrolladores no se animaban a realizar pruebas periódicas de cada modificación si esto implicaba obtener la respuesta horas después.

A pesar de que los desarrolladores deberían lanzar las pruebas localmente para evitar cometer posibles errores y propagarlos a sus colegas, tienden a delegar esta validación a la integración continua. En consecuencia, se realizan menos pruebas a nivel local y se envían los posibles errores a los demás desarrolladores del equipo (Blondeau et al., 2017).⁠

En nuestra experiencia, a medida que se avanza con el proyecto de software, la cantidad y complejidad de las pruebas unitarias aumenta y la potencia de la computadora del desarrollador es casi siempre la misma que al inicio del proyecto, esto incide negativamente en el tiempo de ejecución, haciendo que el desarrollador se vuelva reacio a ejecutar las pruebas localmente y deleguen esta ejecución a la herramienta de integración continua.

Con el objetivo de reducir el tiempo de ejecución de pruebas unitarias de código, el Sistema a desarrollar en el marco de este TFG tendrá las siguientes ventajas:

* Disminuirá el tiempo de ejecución de pruebas unitarias dividiendo la carga de trabajo entre varias computadoras interconectadas de este modo el tiempo que toma la ejecución de las pruebas unitarias de código no estará limitado por la potencia de la computadora del desarrollador.
* Permitirá la ejecución de pruebas unitarias sobre el código fuente descargado del repositorio, así también sobre el código del desarrollador utilizando la misma arquitectura distribuida mencionada en el punto anterior.
* La cantidad de pruebas unitarias a ejecutar en cada computadora involucrada será determinada de manera automática, en comparación a las herramientas de integración continua existentes que requieren la determinación manual de la cantidad de pruebas a ejecutar en cada computadora cuando realizan una ejecución distribuida.

1. MARCO TEÓRICO
   1. CICLO DE VIDA DEL DESARROLLO DEL SOFTWARE

Un ciclo de vida abarca todas las etapas del software, desde su inicio con la definición de requisitos hasta su puesta en marcha y mantenimiento (Ruparelia, 2010)⁠.

El ciclo de vida de desarrollo de software o Software Development Life Cycle (SDLC), es un marco conceptual o un proceso que considera la estructura de las etapas que intervienen en el desarrollo de una aplicación, desde su estudio inicial de viabilidad hasta su despliegue en el campo y su mantenimiento. Existen varios modelos de SDLC (Modelo de ciclo de vida de desarrollo de software o Software Development Life Cycle Model) que describen diversos enfoques del proceso SDLC. Un SDLCM se utiliza generalmente para describir los pasos que se siguen dentro del marco del ciclo de vida. Hay que tener en cuenta que un modelo es diferente de una metodología en el sentido de que el primero describe lo que hay que hacer mientras que el segundo, además, describe cómo hacerlo. Por tanto, un modelo es descriptivo mientras que una metodología es prescriptiva (Ruparelia, 2010)⁠.

En cuanto a la terminología, los términos "proceso (de desarrollo) de software" y "ciclo de vida (de desarrollo) de software" describen dos conceptos estrechamente relacionados pero no idénticos. Mientras que los procesos de desarrollo de software describen los detalles de los pasos individuales de desarrollo, su entrada y salida y cómo combinarlos, un SDLCM describe la combinación y la relación temporal de los distintos procesos (Kneuper, 2017)⁠.

* + 1. Modelos De **Proceso De Software Prescriptivo O Tradicionales**

Los modelos de proceso prescriptivo fueron propuestos originalmente para poner orden en el caos del desarrollo de software. La historia indica que estos modelos tradicionales han dado cierta estructura útil al trabajo de ingeniería de software y que constituyen un mapa razonablemente eficaz para los equipos de software (Pressman, 2010, p. 33)⁠

En éstos modelos se prescriben un conjunto de elementos del proceso: actividades estructurales, acciones, tareas, productos del trabajo, aseguramiento de la calidad y mecanismos de control del cambio para cada proyecto. Cada modelo también prescribe un flujo del proceso (de trabajo), es decir, la manera en la que los elementos del proceso se relacionan entre sí (Pressman, 2010).

El proceso que se adopte mejor al proyecto en la que se está trabajando depende del software que se está elaborando. Un proceso puede ser apropiado para crear software destinado a un sistema de control electrónico de un aeroplano, mientras que para la creación de un sitio web, el proceso seleccionado podría ser distinto (Pressman, 2010)⁠.

A continuación se seleccionaron algunos modelos para ver cómo se componen y funcionan sus diferentes flujos de trabajo, con esto se pretende indicar que las pruebas son parte de prácticamente todos los modelos existentes.

Modelo **en** Cascada

El modelo en cascada, también denominado ciclo de vida clásico, sigue un enfoque sistemático y secuencial para el desarrollo del software, que comienza con la especificación de los requerimientos por parte del cliente y avanza a través de planeación, modelado, construcción y despliegue, para concluir brindando soporte al software terminado (Pressman, 2010)⁠. En este modelo, las pruebas forman parte de la etapa de construcción (Ver 2).

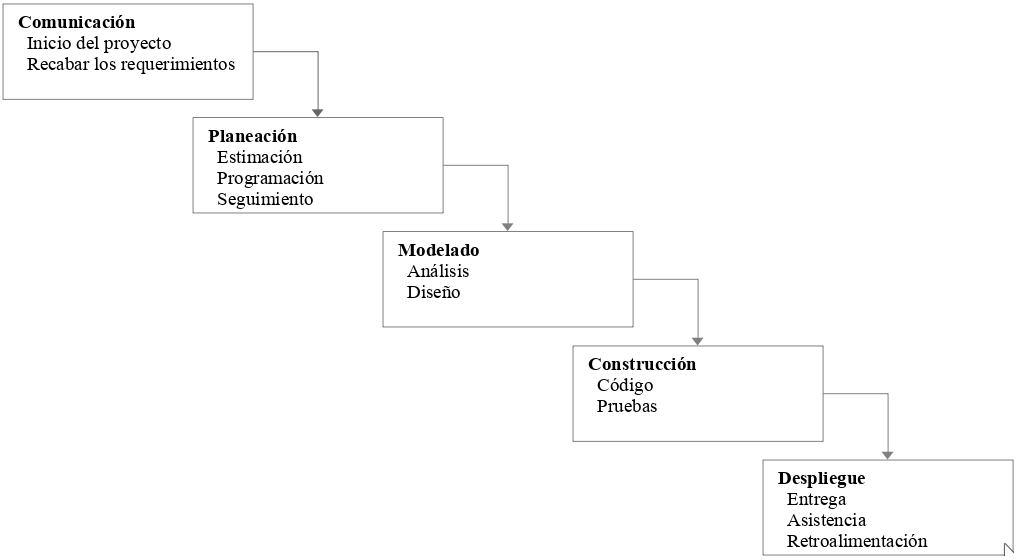


Figura 2: *Modelo en Cascada.*

*Adaptado de Modelo de la cascada (p. 34), por Roger S. Pressman, 2010, McGraw-Hill Interamericana.*

Cataldi, Lage, Pessacq y García Martínez (1999)⁠ menciona las siguientes características del modelo en cascada:

* Cada fase empieza cuando se ha terminado la anterior.
* Para pasar a la fase siguiente es necesario haber logrado los objetivos de la fase previa.
* Es útil para un control de fechas de entregas.
* Al final de cada fase las personas involucradas (como los usuarios y encargados técnicos) tienen la oportunidad de revisar el progreso del proyecto. (p. 187)

Modelo Incremental

Según Pressman (2002)⁠, el modelo incremental combina la forma secuencial e iterativa a través de prototipos funcionales, es decir, cada evolución del proyecto se considera un in cremento (Ver 3).

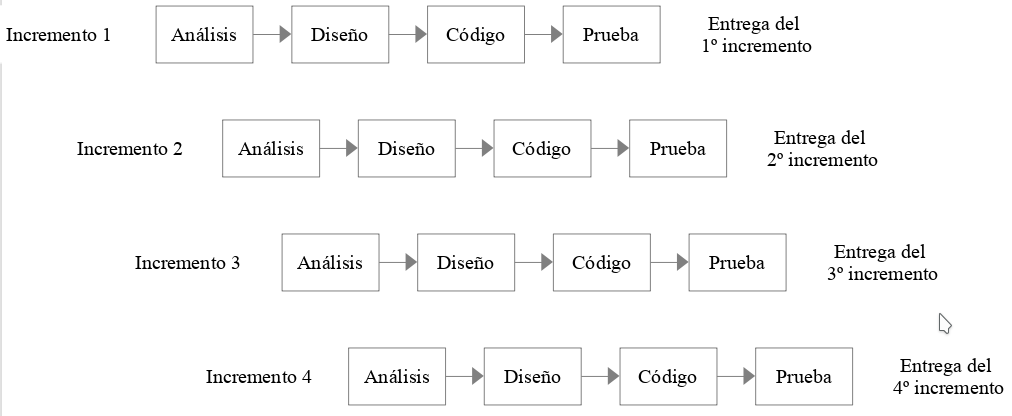


Figura 3: *Modelo Incremental.*

Adaptado de El modelo incremental (p. 24), por Roger S. Pressman, 2002, McGraw-Hill Interamericana.

El primer incremento que sale, contiene elementos básicos y fundamentales del proyecto hasta seguir con los siguientes incrementos para mejorar su funcionalidad, priorizando los requerimientos más importantes para el cliente.

“El modelo incremental se centra en la entrega de un producto operacional con cada incremento. Los primeros incrementos son versiones «incompletas» del producto final, pero proporcionan al usuario la funcionalidad que precisa y también una plataforma para la evaluación” (Pressman, 2002, p. 24)⁠.

Modelo V

El modelo tradicional de desarrollo y pruebas es conocido como el modelo V, donde se describen las actividades de desarrollo en el lado Izquierdo y las correspondientes actividades en pruebas en el lado derecho. El modelo V (Ver 4) plantea que en todo este proceso se debe llevar a cabo la verificación y validación y se realiza como se menciona a continuación. Cada nivel de desarrollo se verifica con el nivel anterior, es decir, se comprueba si los requisitos y definiciones de niveles previos han sido implementados de forma correcta.

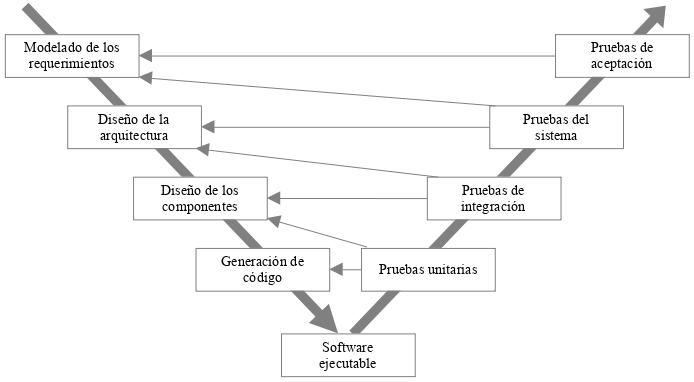


Figura 4: *Modelo V.*

Adaptado de El modelo en V (p. 35), por Roger S. Pressman, 2010, McGraw-Hill Interamericana.

Pressman (2010)⁠ menciona que:

A medida que el equipo de software avanza hacia abajo desde el lado izquierdo de la V, los requerimientos básicos del problema mejoran hacia representaciones técnicas cada vez más detalladas del problema y de su solución. Una vez que se ha generado el código, el equipo sube por el lado derecho de la V, y en esencia ejecuta una serie de pruebas (acciones para asegurar la calidad) que validan cada uno de los modelos creados cuando el equipo fue hacia abajo por el lado izquierdo. (p. 34)

* 1. Pruebas

Las pruebas forman parte de una de las fases del ciclo de de desarrollo de un software, y es la que estaremos profundizando ya que estaremos tratando sobre el mismo en el desarrollo del TFG. Las pruebas forman parte de una de las etapas fundamentales para el aseguramiento de la calidad de un producto. Se puede definir como una actividad en la cual un sistema o uno de sus componentes se ejecutan en circunstancias previamente especificadas. Los resultados de la ejecución se observan y registran con el fin de realizar posteriormente una evaluación de algún aspecto («IEEE Standard Glossary of Software Engineering Terminology», 1990).

* + 1. Pruebas De Software

La prueba es el proceso de ejecutar un programa con la intención de encontrar errores (Myers, Badgett, Thomas y Sandler, 2004)⁠.

(Bertolino, 2001)⁠ menciona que las pruebas de software consisten en la verificación dinámica del comportamiento de un programa en un conjunto finito de casos de prueba, adecuadamente seleccionados del dominio de ejecuciones generalmente infinito, contra el comportamiento esperado especificado.

La calidad en un software es una característica necesaria para que el mismo sea confiable y las pruebas contribuyen a la calidad del software, aunque sea un proceso que requiere de un alto porcentaje de tiempo, pero que a la larga se refleja en ganancia. Estas deben comenzar desde que el desarrollador inicia la construcción del sistema y deben finalizar antes del despliegue del mismo.

Según Mera Paz (2016)⁠ existen 7 principios de las pruebas que permiten establecer unas pautas comunes para que las empresas de desarrollo de software las conozcan y adapten a sus procesos de pruebas. Que si se aplican correctamente dichos principios las empresas tendrán un aseguramiento de la calidad del software. Los principios son los siguientes:

* **Principio Nº 1. Las pruebas demuestran la presencia de defectos:** Las pruebas son unas herramientas que permiten identificar la presencia de defectos; sin embargo, no garantizan que no haya defectos ocultos en el software, y el hecho de que no se identifiquen defectos no es una evidencia de que el software esté totalmente correcto.
* **Principio Nº 2. Las pruebas exhaustivas no existen:** “Probar todo un aplicativo de extremo a extremo con todas las entradas de datos y condiciones es algo imposible”; en vez de tratar de conseguir ese tipo de pruebas se debe realizar un análisis de los riesgos para establecer prioridades y tomar decisiones adecuadas en la utilización de talento humano y recursos, centralizando los esfuerzos en las pruebas.
* **Principio Nº 3. Pruebas tempranas:** Identificar los defectos en etapas tempranas, cuanto más rápido se identifiquen los defectos, más se ahorrará en todo tipo de recursos.
* **Principio Nº 4. Agrupación de defectos:** Las pruebas deben concentrarse de manera proporcional. Por lo general, la mayoría de los fallos operativos se concentran en un número reducido de módulos.
* **Principio Nº 5. Paradoja del pesticida:** Si se repite la misma prueba una y otra vez, la misma serie de casos de prueba dejará de encontrar nuevos defectos; es importante que los casos de prueba se revisen periódicamente y escribir nuevos casos de prueba con el objetivo de encontrar más defectos.
* **Principio Nº 6. Las pruebas dependen del contexto:** Las pruebas dependerán del contexto en el cual se ejecuten; por ejemplo, las que sean para un sistema crítico de seguridad como el financiero, se requiere un mayor número de pruebas en comparación con otras aplicaciones de un nivel de complejidad menor o menos críticos.
* **Principio Nº 7. Falacia de ausencia de errores:** La detección y corrección de los defectos no sirven de nada si el sistema no cumple con los requerimientos o necesidades del usuario. (pp. 168-169)
  + 1. **Métodos De Testeo**
* **Caja Negra:** Es la técnica de prueba sin tener ningún conocimiento del funcionamiento interno de la aplicación. El probador ignora la arquitectura del sistema y no tiene acceso al código fuente. Por lo general, al realizar una prueba de caja negra, un probador interactuará con la interfaz de usuario del sistema al proporcionar entradas y examinar las salidas sin saber cómo y dónde se trabaja con las entradas (tutorialspoint.com, 2011)⁠.
* **Caja Gris:** La prueba de caja gris es una técnica para probar la aplicación con un conocimiento limitado del funcionamiento interno de una aplicación. Dominar el dominio de un sistema siempre le da al probador una ventaja sobre alguien con conocimiento limitado del dominio. A diferencia de las pruebas de caja negra, donde el evaluador solo prueba la interfaz de usuario de la aplicación, en las pruebas de caja gris, el evaluador tiene acceso a los documentos de diseño y la base de datos. Con este conocimiento, el probador puede preparar mejor los datos de prueba y los escenarios de prueba al hacer el plan de prueba (tutorialspoint.com, 2011)⁠.
* **Caja Blanca:** La prueba de caja blanca es la investigación detallada de la lógica interna y la estructura del código. La prueba de caja blanca también se llama prueba de vidrio o prueba de caja abierta. Para realizar pruebas de caja blanca en una aplicación, el evaluador debe tener conocimiento del funcionamiento interno del código. El evaluador debe echar un vistazo al código fuente y descubrir qué unidad/fragmento del código se está comportando de manera inapropiada (tutorialspoint.com, 2011)⁠.
  + 1. Niveles De Pruebas

Pauta Ayabaca y Moscoso Bernal (2017)⁠ mencionan los siguientes niveles de pruebas de software:

* **Pruebas unitarias:** Comprueba si el código funciona y cumple con las especificaciones necesarias para su desempeño óptimo, se focaliza en verificar cada módulo con lo que mejora el manejo de la integración de lo más básico a los componentes mayores.
* **Pruebas de integración:** Comprueba si los componentes funcionan correctamente luego de integrarlos, identificando los errores producidos por la combinación, definiendo si las interfaces entre los usuarios y las aplicaciones funcionan de una manera adecuada.
* **Pruebas de regresión:** Comprueba si los cambios efectuados en una parte del programa afectan a otras partes de la aplicación.
* **Pruebas de humo (Smoke Testing o Ad Hoc):** Comprueba los errores tempranamente revisando el sistema constantemente, lo cual permite disminuir la dificultad en el momento de la integración reduciendo así los riesgos. Con el smoke testing no se busca realizar pruebas a todo el sistema de forma exhaustiva, si no que realizar validaciones seleccionadas sobre las funcionalidades más críticas para luego proceder con pruebas más complejas.
* **Pruebas del sistema:** Están enfocadas directamente a los requisitos tomados de los casos de uso según el giro del negocio, verificando el ingreso, procesamiento, recuperación de los datos y la implementación propiamente dicha. (pp. 30-31). El principal objetivo es que la aplicación cubra las necesidades del negocio, entre las cuales tenemos: prueba de funcionalidad, usabilidad, performance, documentación, procedimientos, seguridad, controles, volumen, esfuerzo (Stress).

En la tabla 3 se puede observar los niveles de pruebas y sus respectivos métodos de testeo.

|  |  |
| --- | --- |
| **NIVEL DE PRUEBA** | **MÉTODO DE TESTEO** |
| Pruebas unitarias | Caja Blanca. |
| Pruebas de integración | Caja Blanca y Negra. |
| Pruebas de regresión | Caja Blanca y Negra. |
| Pruebas de humo | Caja Negra. |
| Pruebas del sistema | Caja Negra. |

**Tabla 3:** Niveles de prueba y sus métodos de testeo.

**Nota:** Esta tabla ha sido adapata de "Guía de Testing de Software: Principales tipos de Pruebas de Software" por A. C. Rocha,2010, p. 1 (https://gtsw-es.blogspot.com/2010/12/principales-tipos-de-testing-de.html)

* 1. Desarrollo dirigido
     1. **Desarrollo Dirigido Por Pruebas De Aceptación (ATDD)**

El desarrollo guiado por pruebas de aceptación (ATDD) es una práctica en la que todo el equipo analiza en colaboración los criterios de aceptación, con ejemplos y luego los proyecta en un conjunto de pruebas de aceptación concretas, antes de que comience el desarrollo. Es la mejor manera para asegurar que todos tengan la misma comprensión compartida de lo que realmente están construyendo (Hendricksons, 2008)⁠.

Al realizar un desarrollo dirigido por pruebas de aceptación, como parte de una práctica ágil, las pruebas se crean de manera iterativa, comenzando antes y continuando durante la implementación de una historia de usuario.

Las historias de usuario deben incluir criterios de aceptación y, a su vez, estos pueden convertirse en (borradores) pruebas de aceptación. Todo esto sucede a través de representantes comerciales, desarrolladores y testers que trabajan juntos en un taller de especificación. El taller no se trata solo de crear pruebas de aceptación. El objetivo real es comprender en colaboración lo que el software debe y no debe hacer (Black et al., 2017)⁠.

* + 1. **Desarrollo Dirigido Por Comportamiento (BDD)**

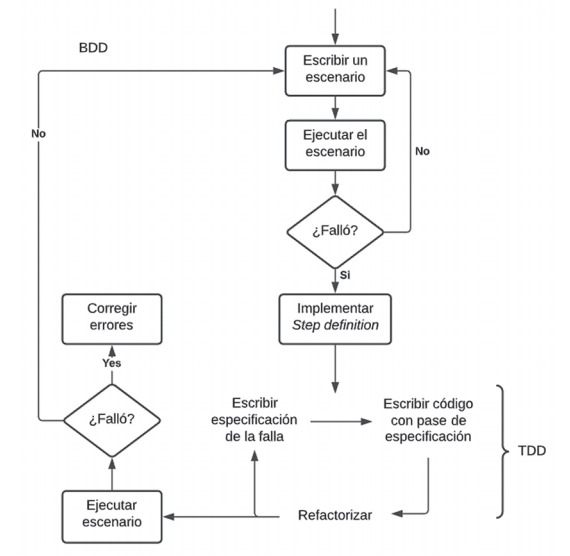
BDD (Behaviour-Driven Development) comienza desde el punto de vista de que los comportamientos del software son más fáciles de entender para las partes interesadas cuando participan en la creación de pruebas. La idea es, junto con todos los miembros del equipo, representantes comerciales e incluso clientes, definir los comportamientos del software. Esta definición puede ocurrir en un taller de especificación, luego, las pruebas se basan en los comportamientos esperados y el desarrollador ejecuta estas pruebas todo el tiempo mientras desarrolla el código (Black et al., 2017)⁠.

Esos quiere decir que en conjunto se debe definir las características que desean ver en el software antes de que el programador empiece a desarrollar las funcionalidades.

Pero lo más importante, BDD proporciona un lenguaje común basado en oraciones simples y estructuradas expresadas en el idioma nativo de las partes interesadas que facilitan la comunicación entre los miembros del equipo del proyecto y las partes interesadas del negocio (Black et al., 2017)⁠.

Uno de los formatos a definir con los colaboradores es el dado/cuando/entonces, con ésto pueden ir definiendo los mismo cómo se deberá comportar el software y así tener un panorama de lo que se espera del software

* Dado (Given): Se especifica el escenario, las precondiciones.
* Cuando (When): Las condiciones de las acciones que se van a ejecutar.
* Entonces (Then): El resultado esperado, las validaciones a realizar.

  
Figura 5: *Estructura de BDD*

* + 1. **Desarrollo Dirigido Por Pruebas (TDD)**

El desarrollo basado en pruebas es un método mediante el cual se crean pruebas unitarias, en pequeños pasos incrementales y, en los mismos pequeños pasos incrementales, se crea el código para cumplir con esas pruebas (Black et al., 2017)⁠.

Estas pruebas unitarias permiten a los desarrolladores de software verificar si su código se comporta de acuerdo con su diseño, tanto mientras desarrollan la unidad, como después de realizar cualquier cambio en la unidad. Esto proporciona un nivel de confianza que lleva a muchos desarrolladores a quedarse con TDD una vez que se han acostumbrado al proceso (Black et al., 2017)⁠.

TDD implica primero escribir una prueba de la funcionalidad de bajo nivel esperada y solo luego escribir el código que ejecutará esa prueba. Cuando pasa la prueba, es hora de pasar a la siguiente pieza de funcionalidad de bajo nivel. A medida que aumenta la cantidad de pruebas y la base de código de forma incremental, también necesita refactorizar el código con frecuencia. La fortaleza de TDD es que solo desarrolla el código mínimo que se requiere para pasar las pruebas unitarias existentes, y solo luego pasa a la siguiente prueba y código (Black et al., 2017)⁠.

* 1. Pruebas Unitarias

Una prueba unitaria procura una "unidad" de código de forma aislada y compara los resultados reales con los esperados. Las pruebas unitarias invocan uno o más métodos de una clase para producir resultados observables que se verifican automáticamente (Olan, 2003)⁠. Por lo general son los desarrolladores lo que implementan las pruebas, ya que es óptimo y necesario conocer el código por la cuál se está trabajando.

(Osherove, 2014)⁠ menciona que las pruebas unitarias deben tener las siguientes propiedades:

* Debe ser automatizado y repetible.
* Debe ser fácil de implementar.
* Debe ser pertinente.
* Cualquiera debería poder ejecutarlo con solo presionar un botón.
* Debe ejecutarse rápidamente.
* Debe ser coherente en sus resultados (siempre devuelve el mismo resultado si no cambia nada entre ejecuciones).
* Debe tener el control total de la unidad bajo prueba.
* Debe estar completamente aislado (se ejecuta independientemente de otras pruebas).
* Cuando falla, debería ser fácil detectar lo que se esperaba y determinar cómo identificar el problema.
  + 1. Librer**ías Para Testeo Unitario En Diferentes Lenguajes**

En la 1 se presentaron algunas librerías o frameworks que permiten implementar pruebas unitarias.

* + 1. Frameworks De Pruebas Unitarias En Ruby

En Ruby tenemos una gama amplia de frameworks para pruebas unitarias, cada una de ellas con sus pros y contras. En la 6 podemos ver el ranking de las librerías más utilizadas en los últimos tiempos.

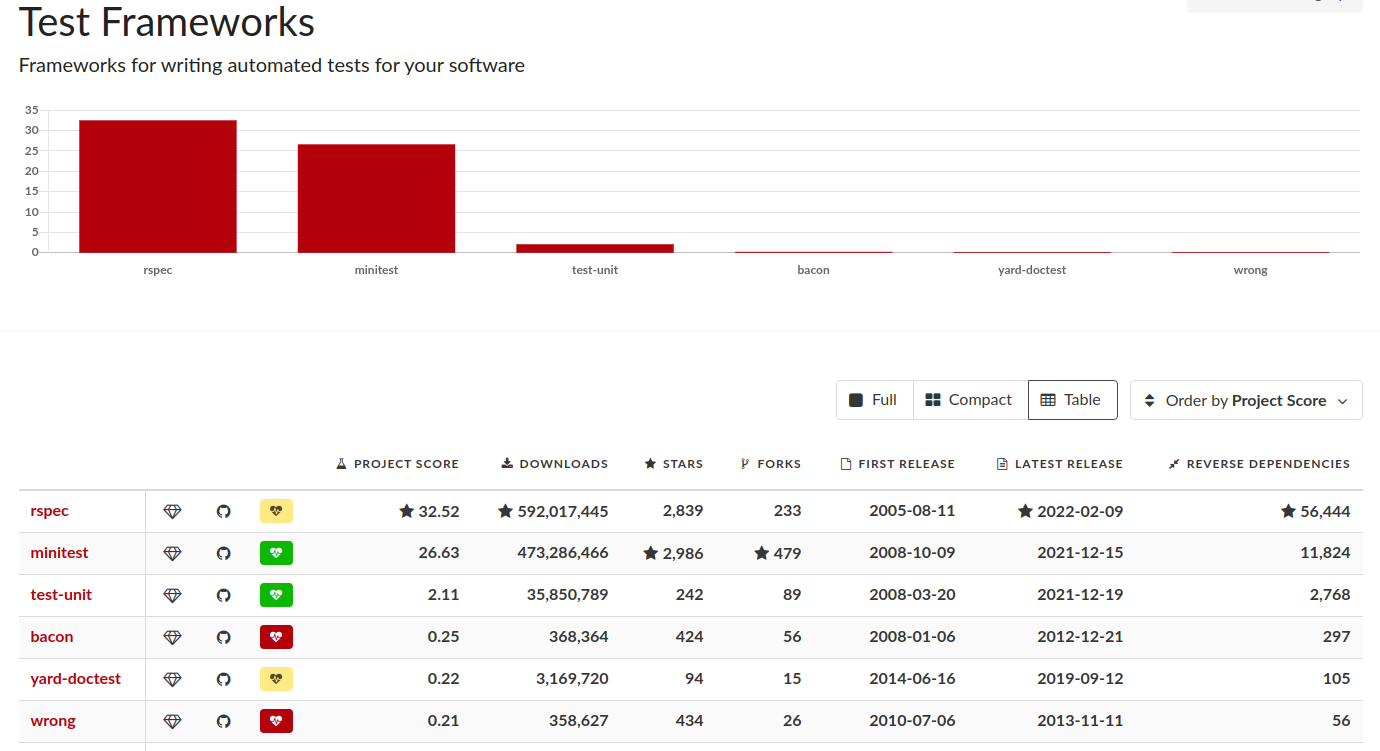


Figura 6: *Ranking de frameworks de pruebas unitarias en Ruby.*

Nota. Datos expresados de acuerdo a la puntuación del proyecto. Test frameworks, 2022. https://www.ruby-toolbox.com/categories/testing\_frameworks?display=table&order=score

* + 1. **Comparativa Entre Rspec Y Minitest**

**MiniTest:** es una herramienta de prueba para Ruby que proporciona [un conjunto completo de instalaciones de prueba](https://semaphoreci.com/blog/automated-testing-cicd).También es compatible con el desarrollo basado en el comportamiento, la simulación y la evaluación comparativa (Davis Ryan, 2022)⁠.

Según (Hourquebie, 2018)⁠:

Minitest es muy simple de utilizar, tiene una sintaxis sencilla y es de rápida implementación con tests de baja a mediana complejidad. Su utilización consiste básicamente en definir clases que se encarguen de testear a otra clase Ruby, o bien a una integración de ellas, dependiendo del tipo de test que se quiere realizar. Cada clase se compone básicamente de tres partes:

* **Test:** Operaciones básicas que se ejecutan para verificar algún requerimiento en particular. Su definición requiere de un string que define aquello que se está testeando.
* **Setup:** Bloque de código que se ejecuta **antes de correr cada test y** una vez por cada uno de ellos, para configurar un entorno común.
* **Teardown:** Bloque de código que se ejecuta **luego de correr cada test**, y una vez por cada uno de ellos. Se utiliza para realizar una limpieza post-test.

**Rspec**: Es una herramienta de desarrollo dirigida por el comportamiento para programadores de Ruby. BDD es un enfoque para el desarrollo de software que combina el desarrollo dirigido por pruebas, el diseño dirigido por dominios y la planificación dirigido por pruebas de aceptación («RSpec», 2022)⁠.

RSpec es, a juzgar por las gemas y proyectos que hemos visto, el framework preferido por los desarrolladores Ruby. Es una herramienta muy robusta y preparada para testear cualquier solución que le haga frente.

La estructura de los tests escritos con RSpec es más flexible que la de Minitest, y permite utilizar diferentes descripciones y contextos para clarificar las incumbencias de aquello que se está testeando.

(Hourquebie, 2018)⁠

* 1. Protocolos
     1. Función De La Capa De Transporte

Según (Cortés, 2008)⁠, la capa de transporte es la encargada de entablar o establecer una sesión de comunicación entre dos softwares, y a su vez poder transmitir datos entre sí. Existen 2 tipos de protocolos de comunicación:

* Protocolo de control de transmisión (TCP)
* Protocolo de datagramas de usuario (UDP)
  + 1. Protocolo De Control De Transmisión (TCP)

El protocolo de control de transmisión ('Transmission Control Protocol', TCP) está pensado para ser utilizado como un protocolo 'host' a 'host' muy fiable entre miembros de redes de comunicación de computadoras por intercambio de paquetes y en un sistema interconectado de tales redes (IETF, 1981)⁠.

Según (Alvarez, 2003)⁠:

TCP (...) utiliza los servicios de IP para su transporte por Internet (...), es un protocolo orientado a conexión. Esto significa que las dos aplicaciones envueltas en la comunicación (usualmente un cliente y un servidor), deben establecer previamente una comunicación antes de poder intercambiar datos. (p. 21)

“TCP conecta un encabezado a los datos transmitidos. Este encabezado contiene múltiples parámetros que ayudan a los procesos del sistema transmisor a conectarse a sus procesos correspondientes en el sistema receptor” (ORACLE, 2010). TCP asegura que cualquier información enviada a través de la red sea recibida por el otro sistema con la cual se estableció la comunicación, lo que lo hace muy confiable en su entrega.

Según (Alvarez, 2003)⁠:

TCP es también un protocolo fiable. La fiabilidad proporcionada por este protocolo viene dada principalmente por los siguientes aspectos:

* Los datos a enviar son reagrupados por el protocolo en porciones denominadas segmentos [Ric98-1]. El tamaño de estos segmentos lo asigna el propio protocolo. (...)
* Cuando en una conexión TCP se recibe un segmento completo, el receptor envía una respuesta de confirmación (Acknowledge) al emisor confirmando el número de bytes correctos recibidos. De esta forma, el emisor da por correctos los bytes enviados y puede seguir enviando nuevos bytes.
* Cuando se envía un segmento se inicializa un temporizador (timer). De esta forma, si en un determinado plazo de tiempo no se recibe una confirmación (Acknowledge) de los datos enviados, estos se retransmiten.
* TCP incorpora un checksum para comprobar la validez de los datos recibidos. Si se recibe un segmento erróneo (fallo de checksum por ejemplo), no se envía una confirmación. De esta forma, el emisor retransmite los datos (bytes) otra vez.
* Como IP no garantiza el orden de llegada de los datagramas, el protocolo TCP utiliza unos números de secuencia para asegurar la recepción en orden, evitando cambios de orden y/o duplicidades de los bytes recibidos.
* TCP es un protocolo que implementa un control de flujo de datos. De esta forma, en el envío de datos se puede ajustar la cantidad de datos enviada en cada segmento, evitando colapsar al receptor. Este colapso sería posible si el emisor enviará datos sin esperar la confirmación de los bytes ya enviados. (pp. 21-22)

Para lograr todos los puntos de fiabilidad citados anteriormente TCP posee una cabecera un poco más complicada (Ver 7).

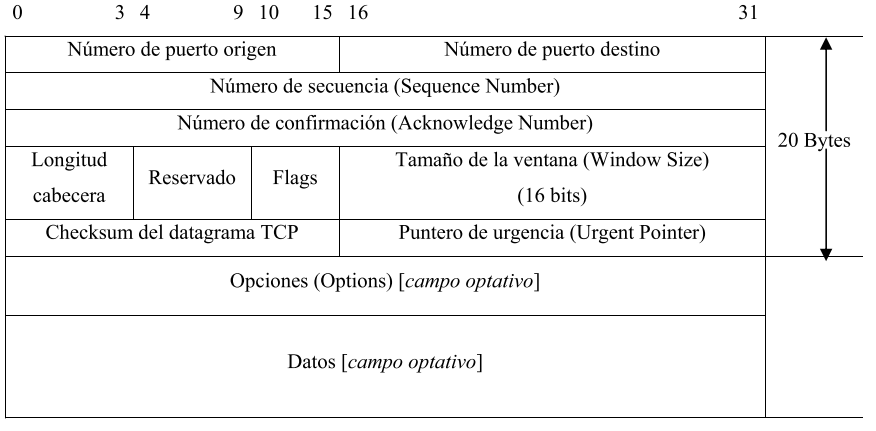


Figura 7: *Cabecera de un segmento TCP.*

Cabecera de un segmento TCP (p. 23), por Gabriel Verdejo Alvarez , 2003, Universitat Autònoma de Barcelona.

* + 1. Protocolo De Datagramas De Usuario (UDP)

El protocolo UDP (Protocolo de datagramas de usuario) proporciona un servicio no orientado a conexión para los procedimientos de la capa de aplicación. Según esto último, es un servicio no seguro, lo cual implica que la entrega y la protección frente a duplicados no están garantizadas.

UDP es un protocolo mucho más simple y no verifica las conexiones entre los hosts transmisores y receptores, lo que significa que no requiere que el host receptor reconozca que ha recibido la información enviada por el host emisor. Esto hace que UDP sea poco confiable, pero funciona bien cuando TCP es demasiado lento o cuando uno o dos paquetes perdidos no marcarán una gran diferencia en el resultado final, como con la transmisión de vídeo, radio y streaming . (Oracle, 2010)⁠

UDP al ser un protocolo simple y no fiable, posee una cabecera mucho más sencilla que la del TCP (Alvarez, 2003)⁠. (Ver 8)

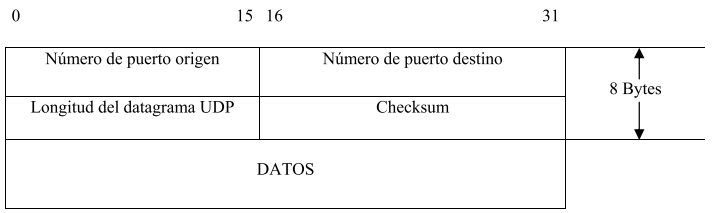


Figura 8: *Cabecera de un datagrama UDP.*

Cabecera de un datagrama UDP (p. 20), por Gabriel Verdejo Alvarez , 2003, Universitat Autònoma de Barcelona.

“El número de puerto (Port) se utiliza en la comunicación entre dos ordenadores para diferenciar las diferentes conexiones existentes. Si tenemos varias comunicaciones desde nuestro ordenador (...), al recibir un datagrama IP debemos saber a cual de las conexiones pertenece” (Alvarez, 2003, pp. 19-20)⁠

“La longitud del datagrama UDP (...) hace referencia al tamaño del datagrama en bytes, y engloba la cabecera (...) más los datos que transporta” (Alvarez, 2003, p. 20)⁠.

El campo de checksum (...), sirve como método de control de los datos, verificando que no han sido alterados. Este checksum cubre tanto la cabecera UDP como los datos enviados. (…) . Si se detecta un error en el checksum, el datagrama es descartado sin ningún tipo de aviso (Alvarez, 2003, p. 20)⁠.

* + 1. Protocolo De Copia Segura (SCP)

La función Protocolo de copia segura (SCP) proporciona un método seguro y autenticado para copiar configuraciones de conmutadores o archivos de imágenes de conmutadores. SCP se basa en Secure Shell (SSH)

* 1. SOCkets

(Fúquene Ardila, 2011)⁠ define al socket (Ver 9) como:

un método para la comunicación entre un programa del cliente y un programa del servidor en una red. Un socket se define como el punto final en una conexión. Los sockets se crean y se utilizan con un sistema de peticiones o de llamadas de función a veces llamados interfaz de programación de aplicación de sockets (...).

Un socket es también una dirección de Internet combinando una dirección IP (...) y un número de puerto (...).

Un socket es un punto final de un enlace de comunicación de dos vías entre dos programas que se ejecutan a través de la red. (p. 49)

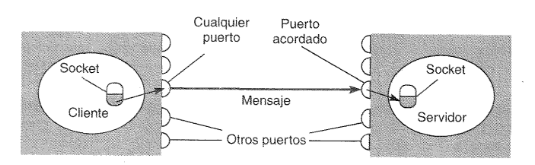


Figura 9: *Comunicación entre sockets.*

Comunicación entre sockets (p. XX), por Fulano de tal , Año, Publicador.

* + 1. Sockets Según Su Orientación
* **Orientado a conexión:** Establece un camino virtual entre servidor y cliente, fiable sin pérdidas de información ni duplicados, la información llega en el mismo orden que se envía. El cliente abre una sesión en el servidor y este guarda un estado del cliente.
* **Orientado a no conexión:** Envío de datagramas de tamaño fijo .No es fiable, puede haber pérdidas de información y también duplicados, la información puede llegar en distinto orden del que se envía. No se guarda ningún estado del cliente en el servidor, por ello, es más tolerante a fallos del sistema. (Fúquene Ardila, 2011, p. 50)⁠
  + 1. Arquitectura Cliente - Servidor

Una arquitectura básica de sockets consiste de 2 programas: Un servidor y un Cliente.

Servidor corresponde a la instancia en que un proceso crea un socket. Cliente corresponde a la instancia en que un proceso se conecta a un socket que se ha creado en otro proceso. Una vez que el socket servidor está habilitado, escucha conexiones y las acepta según su capacidad definida al crearse (Pineda y Méndez, 2019, p. 3).⁠⁠

El servidor es el encargado de responder las peticiones de uno o más clientes (Ver 10)

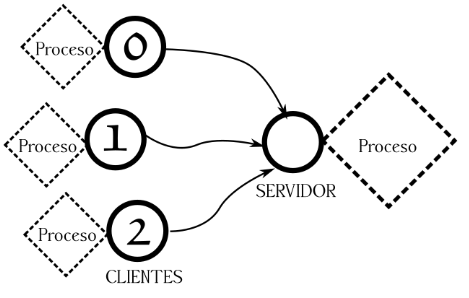


Figura 10: *Ejemplo básico de Cliente-Servidor.*

Ejemplo de Cliente Servidor (p. 3), por David Pineda y Daniel Méndez , 2018, Universidad de Chile.

* 1. DIFUSIÓN DE DATOS
     1. Unidifusión (Unicast)

El proceso de unidifusión es transmitir paquetes de datos al destino. En el momento de reenviar el paquete de datos, este método de transmisión de mensajes usa la dirección de destino en el paquete de datos para buscarlo en la tabla de enrutamiento (Kaur, Singh y Singh, 2016).⁠

Usando la técnica de unidifusión, existe sólo un emisor y un receptor. Si algún emisor necesita o debe enviar un mensaje a varios destinos, tendrá que enviar varios mensajes de unidifusión, cada uno de esos mensajes debe ser dirigido a un destino. Como podemos ver en la 11, los mensajes de unidifusión se enviarán a dispositivos específicos utilizando la dirección IP específica del dispositivo, como dirección de destino en el paquete. (Cicnavi, 2011).

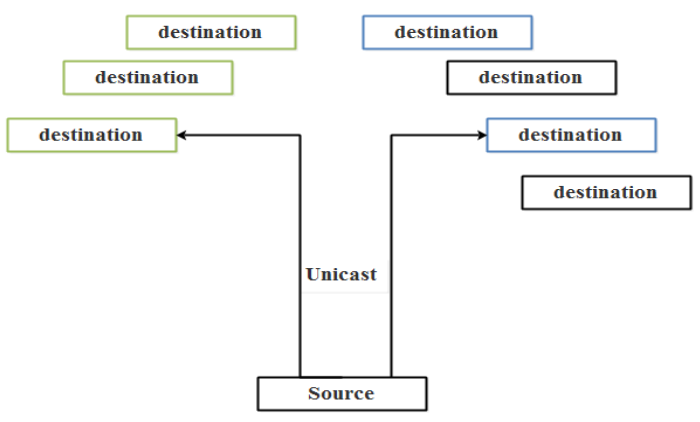


Figura 11: *Envío de paquete, utilizando la técnica Unicast.*

Adaptado de Unicast message Transmission in MANETs (p. 16), por Shivraj Kaur, Kulwinder Singh y Yadvinder Singh , 2016, International Journal of Computer Applications.

* + 1. Multidifusión (Multicast)

Primeramente es importante definir IGMP (Internet Group Managment Protocol) IGMP es un protocolo usado para registrar de forma dinámica cada uno de los hosts de un grupo multicast en una determinada LAN. Los hosts anuncian su pertenencia a grupos enviando mensajes IGMP a su router multicast local. Los routers con IGMP habilitado procesan los mensajes IGMP recibidos, y además periódicamente envían peticiones para descubrir qué grupos están activos o inactivos en una subred en concreto.

Ya que definimos IGMP podemos proseguir con Multicast. Multicast se basa en el concepto de un grupo. Un grupo de multidifusión es un grupo arbitrario de receptores que expresa su interés en recibir un flujo de datos en particular. Este grupo no tiene límites, las físicas o geográficas anfitriones pueden ubicarse en cualquier lugar de Internet o cualquier red interna privada. Los anfitriones que estén interesadas en recibir los datos que fluyen a un grupo en particular deben unirse al grupo utilizando IGMP.

* + 1. **Difusión** (Broadcast)

Broadcast “se utiliza para enviar paquetes a todos los hosts en la red usando la dirección de broadcast para la red” (Cisco, 2022)⁠

Si el paquete tiene una dirección broadcast, todos los dispositivos que reciban ese mensaje lo procesarán (Ver ). Entonces, todos los dispositivos en el mismo segmento de red verán el mismo mensaje (Cicnavi, 2011)⁠.

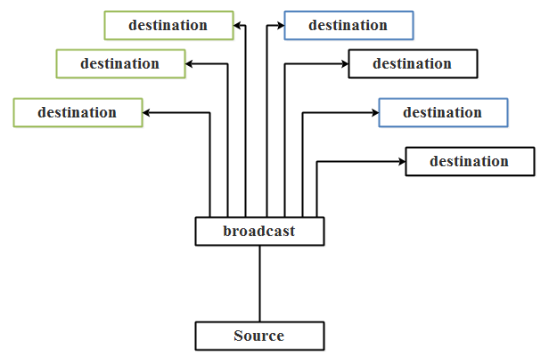


Figura 12: *Envío de paquete utilizando Broadcast.*

Adaptado de Broadcast message Transmission in MANETs (p. 17), por Shivraj Kaur, Kulwinder Singh y Yadvinder Singh , 2016, International Journal of Computer Applications.

1. Conclusi**ón**

Con el uso del sistema y su arquitectura distribuida con computadoras interconectadas entre sí mediante red local utilizando por lo menos dos computadoras, fue posible reducir el tiempo de ejecución y obtención de resultados de las pruebas unitarias.

Referencia Bibliográfica

Alvarez, G. V. (2003). *SEGURIDAD EN REDES IP* (Universitat Autònoma de Barcelona). Universitat Autònoma de Barcelona. Recuperado de https://www.cs.upc.edu/~gabriel/

Athanasiou, D., Nugroho, A., Visser, J., & Zaidman, A. (2014). Test Code Quality and Its Relation to Issue Handling Performance. *IEEE Transactions on Software Engineering*, *40*(11), 1100-1125. https://doi.org/10.1109/TSE.2014.2342227

Bertolino, A. (2001). CHAPTER 5 SOFTWARE TESTING. En A. Abran & J. W. Moore (Eds.), *Guide to the Software Engineering Body of Knoledge SWEBOK* (pp. 69-86). Los Alamitos, California: IEEE Computer Society.

Black, R., Coleman, G., Walsh, M., Cornanguer, B., Forgács, I., Kakkonen, K., & Sabak, J. (2017). *Agile Testings Foundations: An ISTQB Foundation Level Agile Tester guide* (T. C. I. for I. ‎ BCS, Ed.). Swindon, Inglaterra.

Blondeau, V., Etien, A., Anquetil, N., Cresson, S., Croisy, P., Stéphane, D., & Ducasse, S. (2017). What are the Testing Habits of Developers? A Case Study in a Large IT Company. *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 58-68. Shanghai, China: IEEE. https://doi.org/10.1109/ICSME.2017.68

Cataldi, Z., Lage, F., Pessacq, R., & García Martínez, R. (1999). Ingenieria de software educativo. En D. de P. de la F. de I. UBA (Ed.), *V Congreso Internacional de Ingeniería Informática* (pp. 185-199). Recuperado de http://laboratorios.fi.uba.ar/lsi/c-icie99-ingenieriasoftwareeducativo.pdf

Chowdhury, A. E., Bhowmik, A., Hasan, H., & Rahim, M. S. (2017). Analysis of the Veracities of Industry Used Software Development Life Cycle Methodologies. *AJSE*, *16*(2), 75-82. https://doi.org/10.53799/ajse.v16i2.71

Cicnavi. (2011). THE DIFFERENCE BETWEEN UNICAST, MULTICAST AND BROADCAST MESSAGES. Recuperado 28 de marzo de 2022, de Utilize Windows: Information Technology Info website: http://www.utilizewindows.com/the-difference-between-unicast-multicast-and-broadcast-messages/

Cisco. (2022). Introducción a redes: Direcciones de red IPv4. Recuperado 23 de marzo de 2022, de Cisco Networking Academy website: http://itroque.edu.mx/cisco/cisco1/course/module8/#8.1.3.4

Cortés, A. C. (2008). Capítulo 7: Capa de transporte Introducción. Recuperado 27 de marzo de 2022, de Escuela de Ingeniería Electrónica website: http://www.ie.tec.ac.cr/acotoc/CISCO/R&S CCNA1/R&S\_CCNA1\_ITN\_Chapter7\_Capa de transporte.pdf

Davis Ryan. (2022). minitest-5.15.0 Documentación. Recuperado 17 de marzo de 2022, de https://docs.seattlerb.org/minitest/

Fitzgerald, B., & Stol, K. J. (2017). Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*, *123*, 176-189. https://doi.org/10.1016/j.jss.2015.06.063

Fúquene Ardila, H. J. (2011). Implementación cliente servidor mediante sockets. *Revista Vínculos*, *8*(2), 48-59. https://doi.org/10.14483/2322939X.4197

Ganesan, D., Lindvall, M., McComas, D., Bartholomew, M., Slegel, S., Medina, B., … Montgomery, L. P. (2013). An analysis of unit tests of a flight software product line. *Science of Computer Programming*, *78*(12), 2360-2380. https://doi.org/10.1016/j.scico.2012.02.006

Gómez, D., Jústiz, D., & Delgado, M. (2013). Unit Tests of Software in a University Environment. *Computacion y Sistemas*, *17*(1), 69-77. Recuperado de http://www.scielo.org.mx/scielo.php?script=sci\_arttext&pid=S1405-55462013000100008&lng=es&tlng=en

Hendricksons, E. (2008). Acceptance Test Driven Development (ATDD): an Overview. Recuperado 12 de marzo de 2022, de https://testobsessed.com/2008/12/acceptance-test-driven-development-atdd-an-overview/

Hourquebie, L. (2018). Testing en Ruby: Minitest vs RSpec. Recuperado 17 de marzo de 2022, de Unagi website: https://medium.com/unagi/testing-en-ruby-minitest-vs-rspec-683a3d5fab98

IEEE Standard Glossary of Software Engineering Terminology. (1990). *IEEE Std 610.12-1990*, 1-84. https://doi.org/10.1109/IEEESTD.1990.101064

IETF. (1981). *Transmission Control Protocol DARPA Internet Program Protocol Specification*. Recuperado de https://www.rfc-editor.org/rfc/pdfrfc/rfc793.txt.pdf

Kaur, S., Singh, K., & Singh, Y. (2016). A Comparative Analysis of Unicast, Multicast, Broadcast and Anycast Addressing Schemes Routing in MANETs. *International Journal of Computer Applications*, *133*(9), 16-22. https://doi.org/10.5120/ijca2016908001

Kneuper, R. (2017). Sixty years of software development life cycle models. *IEEE Annals of the History of Computing*, *39*(3), 41-54. https://doi.org/10.1109/MAHC.2017.3481346

Leppänen, M., Mäkinen, S., Pagels, M., Eloranta, V. P., Itkonen, J., Mäntylä, M. V., & Männistö, T. (2015). The highways and country roads to continuous deployment. *IEEE Software*, *32*(2), 64-72. https://doi.org/10.1109/MS.2015.50

Mera Paz, J. A. (2016). Análisis del proceso de pruebas de calidad de software. *Ingeniería Solidaria*, *12*(20), 163-176. https://doi.org/10.16925/in.v12i20.1482

Myers, G. J., Badgett, T., Thomas, T. M., & Sandler, C. (2004). *The Art of Software Testing* (2a ed.). New Jersey: John Wiley & Sons.

Olan, M. (2003). Unit testing: test early, test often. *Journal of Computer Science in College*, *19*(2), 319-328. https://doi.org/10.5555/948785.948830

Oracle. (2010). Introducción al conjunto de protocolos TCP/IP (Guía de administración del sistema: servicios IP). Recuperado 12 de marzo de 2022, de Oracle website: https://docs.oracle.com/cd/E19957-01/820-2981/6nei0r0r9/index.html

Osherove, R. (2014). *The Art of Unit Testing* (2a ed.). Shelter Island, NY 11964: Manning Publications Co.

Pauta Ayabaca, L., & Moscoso Bernal, S. (2017). Verificación y Validación de Software. *Killkana Técnica*, *1*(3), 25-32. https://doi.org/10.26871/killkana\_tecnica.v1i3.112

Pineda, D., & Méndez, D. (2018). Socket , Sistema Elemental para la Comunicación Interprocesos. Recuperado 28 de marzo de 2022, de https://www.researchgate.net/publication/333731438\_Socket\_Sistema\_Elemental\_para\_la\_Comunicacion\_Interprocesos

Pressman, R. S. (2002). *Ingeniería del Software Un enfoque práctico* (5a ed.; McGraw-Hill/Interamericana de España, Ed.). Madrid, España.

Pressman, R. S. (2010). *Ingeniería del software Un enfoque práctico* (7a ed.; McGraw-Hill Interamericana, Ed.). México, D. F.

RSpec. (2022). Recuperado 12 de marzo de 2022, de https://relishapp.com/rspec/

Ruparelia, N. B. (2010). Software Development Lifecycle Models. *SIGSOFT Softw. Eng. Notes*, *35*(3), 8–13. https://doi.org/10.1145/1764810.1764814

Shahin, M., Ali Babar, M., & Zhu, L. (2017). Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access*, *5*, 3909-3943. https://doi.org/10.1109/ACCESS.2017.2685629

tutorialspoint.com. (2011). *Software Testing Tutorial*.

Wang, A. (2016). Quality Assurance at Cloudera: Distributed Unit Testing - Cloudera Engineering Blog. Recuperado 14 de julio de 2018, de Cloudera Engineering Blog website: http://blog.cloudera.com/blog/2016/05/quality-assurance-at-cloudera-distributed-unit-testing/

Anexo