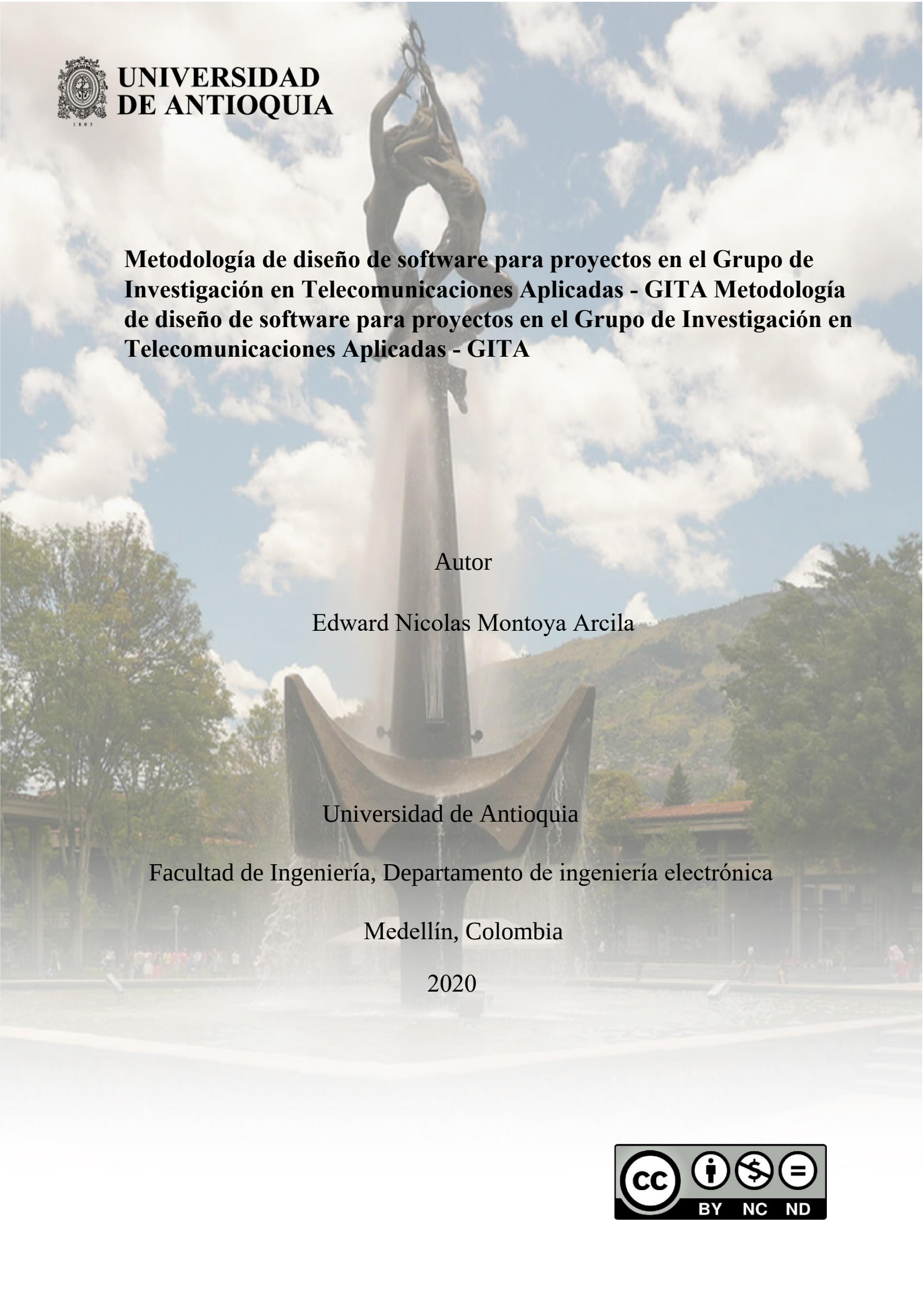

UNIVERSIDAD DE ANTIOQUIA

Metodología de diseño de software para proyectos en el Grupo de Investigación en Telecomunicaciones Aplicadas - GITA Metodología de diseño de software para proyectos en el Grupo de Investigación en Telecomunicaciones Aplicadas - GITA

Autor

Edward Nicolas Montoya Arcila

Universidad de Antioquia

Facultad de Ingeniería, Departamento de ingeniería electrónica

Medellín, Colombia

2020

Tabla de contenido

Tabla de imágenes

Tabla de cuadros

Tabla de códigos

Resumen

Esta tesis tiene como propósito mejorar la calidad de los proyectos de software desarrollados al interior del Grupo de Investigación en Telecomunicaciones Aplicadas (GITA). Para tal fin, en este documento se abordarán los temas de documentación, estructuración y automatización de software.

La temática de la tesis proviene de los constantes problemas que se presentan en mantenibilidad, reusabilidad y organización de código al interior del grupo. Los estándares de calidad con los cuales se presentan los proyectos en la actualidad dificultan su seguimiento y conservación del código con el paso del tiempo. Esta tesis busca solucionar la problemática mencionada proporcionando un marco de referencia estándar que agrupe una serie de buenas prácticas en el campo de la calidad del software. Los resultados finales serán un compendio de entregables que permitirán estandarizar la manera en cómo el grupo de investigación debe implementar los proyectos de software.

Es preciso mencionar que, los principales resultados de esta tesis, solo podrán evidenciarse en el desarrollo de los futuros trabajos del grupo de investigación. Es allí donde se espera que los proyectos presenten mejoras en la comprensión, optimización y en la documentación del código.

Introducción

Según el estándar IEEE 729 [1], la calidad del software se define como: *"Grado con el cual el cliente o usuario percibe que el software satisface sus expectativas"*. Por otro lado, Donald J. Reifer [2], la define como *"El grado en el que un producto de software posee un conjunto especificado de atributos necesarios para cumplir con un propósito enunciado"*. También, Watts Humphrey la define como: *"El producto debe proveer de cierta funcionalidad cuando el usuario la necesite. Si no lo hace, nada de lo demás importa. Segundo, el producto debe trabajar. Si tiene tantos defectos que no funciona con cierta consistencia, el usuario no lo utilizará sin importar otros atributos*". En resumen, hablar acerca de la calidad en el software, es un tema complicado que puede desconcertar, tanto así que evita tratarse en la mayoría de los claustros académicos. Por ejemplo, en la Universidad de Antioquia, la mayoría de sus programas presentan asignaturas que permiten a los estudiantes obtener magnificas bases en programación y uso de software. Sin embargo, la formación en el campo de calidad de software es precaria.

La definición de calidad parece estar en todos lados, cada artículo o proyecto remarca su importancia. Por ejemplo, en la academia existen planes de estudio certificados por su alta calidad. Sin embargo, en su mayoría dejan el concepto la calidad de software como una definición aparte, como un complemento que el profesional debe adquirir en el futuro. En contraste, en la industria, la calidad de software no ha parado de evolucionar. Temas como la estandarización de software y los marcos de referencia vienen avanzando vertiginosamente, dando un sello de calidad a los productos que genera confianza, identidad y es sinónimo de éxito.

En el desarrollo tecnológico acelerado que venimos percibiendo en los últimos años, la estandarización de software se ha convertido en eje estructural del cambio. En el ámbito comercial, la diversificación y crecimiento de la tecnología ha causado que los sistemas queden obsoletos en tiempos muy cortos, dando mayor peso a las decisiones de arquitectura y estandarización para mantener sistemas capaces de reinventarse en el tiempo. La estandarización de procesos tiene como objetivo definir lineamientos eficientes que permitan evitar reprocesos, reducir la variabilidad y generar estructuras funcionales que puedan reutilizarse.

Esta temática ha adquirido tal importancia, que grandes operadores como Google y Amazon han comenzado un proceso de mantenibilidad y calidad en sus procesos de software. Blogs como "Testing Blog" By Google, llevan años generando estándares de calidad en el proceso de diseño, implementación y codificación del software. Incluso estos blogs, han permitido crear comunidades gigantescas que dan soporte y estructuran código Open Source desde todas partes del mundo.

En los ambientes académicos, a pesar del gran rigor con que evalúan su trabajo, aún se continúa con metodologías anticuadas en el desarrollo de software. Por otro lado, en el nicho industrial, se ha comenzado a estandarizar procesos, iniciando una carrera por generar software de calidad que los agilice. En este contexto, los ambientes académicos aún no generan en su núcleo una cultura informática que influencie a sus participantes en metodologías que permitan: estructurar

proyectos de manera idónea, levantar requisitos, adaptarse a una arquitectura común, mantener buenas prácticas y definir correctamente pruebas unitarias.

La Universidad de Antioquia ha comenzado un proceso de actualización que busca mejorar la calidad en sus proyectos. Las certificaciones de calidad se han convertido en una meta que es impulsada desde directrices universitarias. Sin embargo, en ciertos ámbitos aún se tiene un desarrollo incipiente que, lejos de ser generalizado, solo es adoptado por ciertos nichos que por competitividad necesitan seguir los estándares propios de la industria.

En el Grupo de Investigación en Telecomunicaciones Aplicadas (GITA), el crecimiento del uso de software y el aumento de proyectos en los últimos años ha generado dificultades de productividad y seguimiento. El objetivo de este trabajo de grado es diseñar un marco de estandarización que permita dar lineamientos en el uso de buenas prácticas de desarrollo de software, a través del uso de herramientas como GitLab, Jenkins, Wiki, Docker y SonarQube. De manera que sea posible generar una arquitectura común y modelo estandarizado para entregables de software al interior del grupo.

Objetivos

Objetivo general: Desarrollar una metodología que permita apoyar la implementación de proyectos de software en el grupo de investigación GITA.

Objetivos específicos:

- Definir lineamientos para promover estándares de calidad en el desarrollo del software de los proyectos del grupo de investigación.
- Proponer guías que permitan estandarizar los procesos de documentación, control de versiones, organización y pruebas unitarias del software a desarrollar.
- Implementar las herramientas para la gestión de software OpenSource que mejor se adapten a las dinámicas del grupo de investigación GITA.
- Realizar una prueba piloto que permita evaluar los beneficios de la metodología propuesta.

1. Marco Teórico

1.1. Calidad

La necesidad de pensar en gestionar los procesos mediante estándares de calidad surge con la innovación. Mientras más competencia se visualiza en los mercados mundiales, las diferentes compañías buscan maneras de crecer y mantener sus mercados. De allí se puede definir el concepto de calidad como la balanza que determina la elección de un producto sobre otro. En la actualidad conocemos como Gestión de Calidad al conjunto de acciones, medidas y soluciones orientadas a la mejora continua de los procesos internos de una organización, tomando como objetivo principal el aumento del nivel de satisfacción de un grupo de clientes o consumidores [3]. Esta definición, que podría considerarse como la más aceptada en el campo industrial, a su vez es muy limitada lo cual impide que exista una implementación real de la misma.

Dentro de las diferentes definiciones sobre gestión de la calidad, la norma ISO 9001 [4] dice: *"Gestión de la Calidad comprende el conjunto de acciones, planificadas y sistemáticas, que son necesarias para proporcionar la confianza adecuada de que un producto o servicio va a satisfacer los requisitos dados sobre la calidad"*.

Dentro de los autores más conocidos, Feigenhaum [5] define la Gestión de Calidad como *"el conjunto de las características del producto de marketing, ingeniería, fabricación y mantenimiento a través del cual el producto satisface las expectativas del cliente"* (p. 274). Por su parte, Juran [6] plantea *"La calidad significa aquellas características del producto que se ajustan a las necesidades del cliente"*.

1.1.1. ¿Por qué es importante la calidad?

La gestión de calidad de un proyecto es uno de los últimos pasos que se analiza cuando este se estructura y ejecuta, y en algunos casos ni siquiera es un tema para tratar. Sin importar el ámbito del proyecto, la calidad es un tema que durante mucho tiempo se ha dejado de lado. Dejar de lado la calidad significa abandonar el objetivo principal del mercado, el cual es permitir que el cliente se encuentre satisfecho con el producto adquirido. Si esto no se logra no se puede hablar de calidad y a su vez no se puede hablar de crecimiento financiero.

1.1.2. Calidad en el software.

El tema de calidad en el software durante años ha sido enfocado en un conjunto de metodologías que permitan organizar manera estructurada el desarrollo del código. Las metodologías de desarrollo de software han sido el enfoque principal para las compañías, pues éstas brindan

pautas claras para responder a las necesidades del mercado, dando mayor calidad al código y brindando a los clientes productos reusables y mantenibles.

Las grandes casas de software durante años han puesto mucho cuidado en los detalles de su código. Sin embargo, la llegada de internet y la "democratización" del desarrollo de software, han causado que miles de nuevos programadores publiquen su software sin restricciones. Este comportamiento, aparentemente es un gran avance, pero implica tener cientos de librerías OpenSource sin el debido control o garantías de calidad. Este comportamiento ha provocado que, en los últimos años, la comunidad OpenSource haya implementado estándares para mitigar el problema de calidad. De aquí que sea muy importante que, en todo nicho donde se esté desarrollando software, se implementen metodologías de desarrollo y se incentive el uso de estándares de calidad.

1.2. Lenguajes de programación

Un lenguaje de programación es un cálculo matemático o lenguaje formal. Su objetivo es expresar algoritmos de manera inequívoca para personas y máquinas. Como cualquier cálculo, un lenguaje define tanto la sintaxis como la semántica. La sintaxis es la gramática del lenguaje, la notación; la semántica es el significado de esa notación. Los lenguajes de programación se utilizan para facilitar la comunicación sobre la tarea de organizar y manipular información, y para expresar algoritmos con precisión. Sin embargo, es preciso mencionar que algunos autores restringen el término "lenguaje de programación" a aquellos lenguajes que pueden expresar todos los algoritmos posibles [7].

1.2.1. Tipado.

Un lenguaje de programación puede expresar un algoritmo de manera sencilla o complicada según como fue diseñado. Es decir, según las restricciones definidas en su diseño, un algoritmo puede ser en mayor o menor medida más complicado de implementar. Una de las restricciones más comunes en el diseño de lenguajes de programación es el tipado. El sistema de tipos es una herramienta para restringir un lenguaje, los tipos asocian metadatos con valores y las variables que pueden vincularse a ellos.

Un programa bien escrito es aquel en el que se satisfacen las restricciones sobre los metadatos impuestos por el lenguaje y el programa en sí. Cuando se violan, por ejemplo, al asignar un valor String a una variable int en Java, el programa es incorrecto. Algunos de los errores en programa pueden detectarse mediante análisis estático, lo que significa examinar el programa sin ejecutarlo. Otros no pueden detectarse de manera eficiente través del análisis estático, o son estáticamente indecidibles. Sin embargo, muchos de estos errores si pueden detectarse por análisis dinámico, lo que significa ejecutar verificaciones de tipo en tiempo de ejecución, mientras se ejecuta el programa [7].

1.2.2. Lenguajes imperativos y funcionales.

La disciplina de la informática surgió de las matemáticas, en gran parte debido al trabajo de Church y sus alumnos, particularmente Turing. Church y Kleene crearon un sistema matemático llamado cálculo lambda, que trata las funciones matemáticas como valores de primera clase dentro de las matemáticas. Turing creó "la máquina de Turing", una máquina para realizar simples cálculos operativos que, luego, se demostró, que eran equivalentes a modelos mínimos de computación. El modelo de máquina de Turing conduce a una programación imperativa, que opera mutando (cambiando) el estado y avanza por iteración. Los lenguajes Java y C ++ fomentan este estilo. Por su parte, el modelo matemático de Church conduce a la programación funcional, que opera invocando funciones y procede por recursividad. Los lenguajes Scheme, ML, Unix Shell Commands y Haskell fomentan este estilo. Por otro lado, también existen lenguajes que fomentan la combinación de los dos estilos mencionados (imperativo y funcional). Estos lenguajes, conocidos como lenguajes de script (por ejemplo, Perl y Python), favorecen la concisión en todas las expresiones.

En esta tesis se enfocará en los lenguajes de programación imperativos, más específicamente, los que usan el paradigma de programación orientado a objetos.

1.2.2.1. Paradigma de programación orientado a objetos (POO)

La programación orientada a objetos, tal como su nombre lo indica, busca implementar entidades del mundo real como: herencia, privacidad, abstracción, ocultamiento y polimorfismo en programación. El objetivo principal de la programación orientada a objetos consiste en enlazar los datos y las acciones que realiza un programa para brindar flexibilidad al código [8]. Los conceptos claves de la programación orientada a objetos son:

- Clase: Es un tipo de datos definido por el usuario, que contiene atributos y funciones propias, a los que se puede acceder y usar creando una instancia de esa clase. Una clase es como un plano de un objeto, el cual puede ser construido en cualquier momento.
- Objeto: El objeto es una entidad identificable con algunas características y comportamiento. Un objeto es una instancia de una clase. Cuando se define una clase, no se asigna memoria, pero cuando se instancia (es decir, se crea un objeto) se asigna memoria.
- Encapsulación: En términos normales, la encapsulación se define como la recopilación de datos e información en una sola unidad. En la programación orientada a objetos, la encapsulación se define como la unión de los datos y las funciones de una entidad.
- Abstracción: La abstracción de datos es una de las características más importantes de la programación orientada a objetos. La abstracción significa mostrar solo información esencial y ocultar los detalles.

- Polimorfismo: La palabra polimorfismo significa tener muchas formas. En palabras simples, podemos definir el polimorfismo como la capacidad para mostrarse en más de una forma.
- Herencia: Es la capacidad de una clase para derivar propiedades y características de otra clase. La herencia es una de las características más importantes de la programación orientada a objetos.

1.2.3. Propósito de los lenguajes programación.

Un propósito destacado de los lenguajes de programación es proporcionar instrucciones a una computadora. Como tal, los lenguajes de programación difieren de la mayoría de las formas de expresión humana en que requieren un mayor grado de precisión e integridad. Cuando se usa un lenguaje natural para comunicarse con otras personas, los autores y hablantes humanos pueden ser ambiguos y cometer pequeños errores, y aun así esperar que se entienda su intención. Sin embargo, las computadoras hacen exactamente lo que se les dice que hagan, y no pueden entender el código que el programador pretendía escribir. La combinación de la definición del lenguaje, el programa y las entradas del programa deben especificar completamente el comportamiento externo que ocurre cuando éste se ejecuta [9].

Muchos lenguajes de programación han sido diseñados desde cero, otros modificados para satisfacer nuevas necesidades, algunos han sido combinados con otros lenguajes y casi en su mayoría cayeron en desuso. Nunca se ha podido diseñar un lenguaje definitivo que solucione todas las necesidades, aunque existieron intentos de diseñar un lenguaje de computadora "universal" que sirva para todos los propósitos. La necesidad de diversos lenguajes de computadora surge de la diversidad de contextos en los que se usan los lenguajes de programación.

1.2.4. Java.

Es un lenguaje de programación creado en 1995. Su dueño es Oracle y alrededor de 3 billones de dispositivos lo pueden ejecutar [10]. Es usado para:

- Aplicaciones móviles
- Aplicaciones de escritorio
- Aplicaciones web
- Servidores web
- Conexión a bases de datos
- Juegos

1.2.5. Python.

Es el lenguaje de programación más popular del mundo a la fecha. Fue creado por Guido van Rossum y liberado en 1997. La versión principal más reciente de Python es Python 3, sin embargo, Python 2, aunque no se actualiza con nada más que actualizaciones de seguridad, sigue siendo bastante popular. Python fue diseñado para facilitar la lectura y tiene algunas similitudes con el idioma inglés con influencia de las matemáticas. Python usa nuevas líneas para completar un comando, a diferencia de otros lenguajes de programación que a menudo usan punto y coma o paréntesis. Python se basa en la sangría, utilizando espacios en blanco, para definir el alcance; como el alcance de bucles, funciones y clases. Otros lenguajes de programación a menudo usan llaves para este propósito [11].

1.2.6. Comparativa entre Java y Python

Característica\ Lenguaje	**Java**	**Python**
Plataforma	Multiplataforma	Multiplataforma
Sintaxis	Compleja, similar a C/C++, implica más líneas de código.	Simple, similar al inglés, implica menos líneas de código.
Interpretado	Si, mediante compilador	Si, mediante JVM
Paradigmas implementados	POO, paradigma funcional y paradigma de eventos.	POO y paradigma funcional.
Curva de aprendizaje	Alta	Media
Separación de estructuras de programación	Llaves	Sangría
Separación de sentencias	Punto y coma	Salto de línea
OpenSource	Si	Si
Aplicaciones más comunes	Aplicaciones web empresariales, big data, redes de alto desempeño y aplicaciones de escritorio.	Ciencias de datos, machine learning, scripting, aplicaciones web y de robótica.
Uso según TIOBE (The Importance Of Being Earnest)	16.73% de los proyectos en la web	9.32% de los proyectos en la web

Cuadro 1. Comparativa entre Python y Java

1.3. Metodologías de desarrollo de software

Una metodología de desarrollo de software es una forma de gestionar un proyecto de software que permite dar solución a problemáticas como: cuándo se lanzará el software, quién trabaja en qué y qué pruebas se realizan. Ninguna metodología es mejor para todas las situaciones, incluso el método de cascada muy difamado en algunos sectores es apropiado para algunas situaciones. En la práctica, cada organización implementa su gestión de proyectos de desarrollo de software de una manera diferente, incluso dentro de la misma organización, a menudo se utilizan distintas metodologías entro un proyecto y otro.

El objetivo final de una metodología implica medir en términos de costo, el cumplimiento de plazos, la satisfacción del cliente y la solidez del software. Debido a esto, elegir correctamente una metodología puede ser el punto de inflexión para el éxito o el fracaso de un proyecto [12].

1.3.1. Cascada.

Es la metodología más conocida del enfoque tradicional del desarrollo de software, utiliza un método lineal en el que hay un gran énfasis en recopilar requisitos y diseñar la arquitectura del software antes de realizar el desarrollo y sus pruebas. Las ventajas de este tipo de desarrollo radican en que: el proyecto está bien planificado, los costos a medio plazo se disminuyen y los proyectos tienden a estar bien documentados. Sin embargo, las desventajas de esta metodología radican en que es muy difícil ajustar los requisitos del proyecto en medio del desarrollo, lo que a menudo ocurre cuando se descubren problemas o las necesidades cambian. Las desventajas se traducen en lanzamientos de versiones principales con un número significativo de nuevas características en largos periodos de tiempo [12].

1.3.2. Evo.

Es un sistema de desarrollo antiguo y menos conocido, desarrollado en Hewlett Packard. Tiene una similitud significativa con el desarrollo ágil y agrega un enlace a los ciclos de ventas y fabricación. A diferencia de las metodologías agiles, pone más énfasis en tener un gerente técnico para asignar tareas [12].

1.3.3. Familia de procesos unificados.

Es un proceso de desarrollo interactivo para equipos de desarrollo más grandes, a menudo más burocráticos. Los puntos clave de esta metodología residen en: un fuerte enfoque para los casos de uso, que a su vez sugieren requisitos. También hay un énfasis en elegir la mejor arquitectura, las tareas de mayor riesgo se realizan primero para proporcionar un punto de interrupción temprano en el que el proyecto puede cancelarse si está condenado al fracaso. En definitiva, el grupo de trabajo usa de manera eficiente los recursos para lograr realizar porcentajes de requisitos, diseño, implementación y pruebas en paralelo [12].

1.3.4. PRINCE2.

Es el formato de gestión de proyectos ordenado por el gobierno del Reino Unido para proyectos públicos, es común en Europa y no es específico para el desarrollo de software. PRINCE2 se enfoca en el proceso de cómo se deben hacer las cosas a través de siete principios: justificación comercial continua, aprender de la experiencia, roles y responsabilidades definidos, gestionar por etapas, gestionar por excepción, centrarse en productos y adaptarse al entorno del proyecto. Este proceso enfatiza la gestión de recursos y es apropiado para proyectos pequeños [12].

1.3.5. Project Management Body of Knowledge.

Es un marco genérico de gestión de proyectos establecido por el Project Management Institute (PMI) y adoptado como estándar por la American National Standars Institute (ANSI) y el Institute of Electrical and Electronics Engineers (IEEE). Se centra más en lo que se debe hacer (herramientas y técnicas). Este se usa como marco para el desarrollo de software cuando el cumplimiento de esos estándares es un requisito obligatorio [12].

1.3.6. Microsoft Solutions Framework (MSF).

Esta es una metodología genérica de gestión de proyectos centrada en tecnología e innovación (TI), que incluye el desarrollo de software y la implementación de equipos. MSF usa dos modelos: un modelo de equipo que describe los roles de los individuos en el grupo de desarrollo de software, y un modelo de gobierno que describe las etapas del proceso de desarrollo. MSF contiene plantillas para la integración del modelo de madurez ágil y de capacidad [12].

1.3.7. Crystal Methods Methodology.

Los métodos de cristal son una familia de metodologías relacionadas, cada una nombrada con un color. La filosofía de los métodos de cristal propone que el proceso debe diseñarse en torno a las fortalezas y debilidades de las personas en el equipo. Los diversos niveles del método de cristal varían en cuanto a su peso: los métodos livianos pueden ser los mejores para proyectos pequeños y a corto plazo, los métodos de gran peso son más adecuados para casos en los que la vida humana está involucrada, por lo tanto, no se pueden tolerar errores. Todos incorporan entregas frecuentes, comunicación cercana con usuarios expertos y seguridad personal [12].

1.3.8. Joint Application Design (JAD).

Este es un proceso para la recopilación de requisitos en el que varias partes, incluidos varios miembros de la organización del cliente, pasan días participando en un taller de recopilación de requisitos. También se conoce como desarrollo de aplicaciones conjuntas. La observación empírica ayuda a proveer un conjunto de requisitos mejor definido por adelantado, esta metodología proporciona una modesta reducción en los costos de desarrollo [12].

1.3.9. Lean Development.

Este proceso está destinado a dar el costo mínimo absoluto. Los desarrolladores se centran en el 80% hoy, no en el 100% mañana. Tiene una serie de desventajas que van desde mala calidad, características mínimas, y software rápido y barato [12].

1.3.10. Systems Development Life Cycle (SDLC).

Es un proceso más formalizado para manejar grandes proyectos donde la documentación, capacitación, integridad y seguridad son vitales para el éxito del proyecto. Los proyectos SDLC suelen utilizar análisis y diseño orientado a objetos. Se prepararán múltiples modelos para casos de uso, datos relacionales, interfaz de usuario y un modelo conceptual más abstracto [12].

1.3.11. Metodologías ágiles.

Las metodologías ágiles están diseñadas para adaptarse a requisitos cambiantes, minimizar los costos de desarrollo y aun así ofrecer un software de calidad razonable. Los proyectos ágiles se caracterizan por muchos lanzamientos incrementales, cada uno generado en un período de tiempo muy corto. Por lo general, todos los miembros del equipo están involucrados en todos los aspectos de planificación, implementación y pruebas. Además, hay un fuerte énfasis en las pruebas a medida que se escribe el software.

El propósito de este trabajo de grado toma como referencia las metodologías ágiles y los beneficios que esta provee para mejorar la calidad del software.

1.4. Metodologías ágiles

En las metodologías de desarrollo tradicionales, conforme pasaron los años, se visualizó un deterioro considerable para acoplar los nuevos proyectos a las dinámicas de mercado. Mientras los proyectos crecían en escala y complejidad, las metodologías de desarrollo tenían retos titánicos para cumplir objetivos y eficiencia a niveles nunca antes vistos.

Una manera de referirse a las metodologías ágiles es Agile. Esta metogología nace como respuesta a la alta demanda de software, proponiendo un proceso de desarrollo en el que los resultados y entregas son constantes, dando respuestas inmediatas a los problemas innatos del desarrollo. Este proceso fue descrito en el manifiesto ágil [13], un documento que consta de 12 principios diseñados por el creador del concepto.

Sus ventajas son:

- Entregas constantes
- Mayor iteración y pruebas
- Menos errores finales

- Mayor cantidad de pruebas implementadas
- Trabajo en equipo en todo el proceso

Agile permite ahorrar tiempo y dinero, así como eliminar documentación innecesaria. Los requisitos en Agile varían con el tiempo y son un punto claro en cada etapa del desarrollo, lo que ayuda a tener mucha más claridad del objetivo de cada entrega, enfocándose más en la aplicación que en la documentación. Dado que es iterativo en su forma, tiende a tener una retroalimentación regular del usuario final, de modo que la etapa productiva llegue lo antes posible. Agile ayuda a las organizaciones a reducir significativamente el riesgo general asociado con el proceso de desarrollo y garantizar que se maximice el valor comercial [14].

Una de las principales implementaciones de Agile es DevOps (Development and Operations). Esta implementación, mediante un proceso claro y estructurado, definió una serie etapas enmarcadas en los conceptos de integración y entrega continuas, que permiten acelerar el ciclo de desarrollo de una aplicación manteniendo altos estándares de calidad. En resumen, DevOps se ha convertido en el principal referente de las metodologías ágiles a nivel mundial por su flexibilidad, eficiencia y velocidad en el ciclo de vida del desarrollo de software.

1.4.1. Integración y entrega continuas (CI / CD).

La adaptación de las prácticas ágiles permite flexibilidad, eficiencia y velocidad en el ciclo de vida del desarrollo de software (SDLC, Software Development Life Cycle). Según el manifiesto ágil [13], uno de los puntos claves de las metodologías, es la existencia de un flujo de trabajo que permita desarrollar software rápidamente. Sin embargo, en el manifiesto este concepto solo se presentó de manera genérica y posteriormente fue definido como CI/CD.

En el año 2006, [15] Martin Fowler presentó la idea de Integración Continua (CI) como *"una práctica de desarrollo de software donde los miembros de un equipo integran su trabajo frecuentemente mediante herramientas de automatización"*. Más tarde J.Humble y D.Farley [16] extendieron estas ideas al enfoque de la Entrega Continua (CD), la cual definieron como *"la capacidad de obtener cambios de todo tipo en producción, de forma segura, rápida y sostenible"*.

Los principales beneficios de las prácticas de CI/CD son: [17]

- **Facilidad para integrar los cambios realizados por diferentes desarrolladores:** Anteriormente los desarrolladores debían crear módulos enteros por aparte. Esta realidad dificultaba integrar módulos del proyecto, generado retrasos. Sin embargo, la integración continua permite mantener el código actualizado en cada lanzamiento del proyecto.

- **Realimentación del estado del proyecto:** Un cambio en el código puede causar grandes estragos para el proyecto. Esto implica que los comentarios y la visibilidad de los cambios, sean un punto clave para el avance del proyecto. Entender cuándo, cómo y quién realizo un commit con problemas puede garantizar una rápida solución.
- **Aumento de la transparencia y visibilidad**: Dado que los procesos de integración y entrega continuas son bucles, todo el equipo conoce el estado del proyecto en cada etapa.
- **Detección de errores temprana:** Las pruebas unitarias y de integración cumplen un papel trascendental en los procesos de calidad de software. Gracias a este tipo de pruebas el código se mantiene verificado y es posible evitar cadenas de errores que pueden presentarse a lo largo del proyecto.
- **Aumento de la calidad y velocidad de desarrollo:** Cuanto más fácil es probar algo, más fácil es determinar su calidad. La capacidad de prueba se puede caracterizar por cuán controlable, observable, sin errores y descomponible es el software.

1.4.2. Etapas del proceso de integración y entrega continua

El éxito de DevOps como metodología ágil posicionó a su flujo de trabajo como la implementación estándar de las etapas que deberían seguirse en la integración y entrega continua. Las etapas de CI/CD buscan garantizar la creación de aplicaciones de manera segura, rápida y confiable mediante procesos automatizados.

Como lo muestra la Figura 1, las etapas de integración y entrega continua son:

- **Planificación:** Esta etapa hace referencia al proceso de definición y estructuración del proyecto antes del desarrollo de código.
- **Código:** Esta etapa hace referencia a la construcción y gestión del código. En esta etapa se hace uso del software de control de versiones.
- **Construcción:** En esta etapa se realiza el proceso de descarga de dependencias y empaquetado. Para el lector puede ser confuso que las dependencias se descarguen después de escribir el código, pero solo hasta que el código comienza el proceso de construcción es que realmente se genera el archivo ejecutable.
- **Pruebas:** En esta etapa el servidor de automatización prueba el código y verifica que los estándares definidos por los administradores se cumplan.
- **Lanzamiento:** En esta etapa se realiza el empaquetado final del código y se guarda un ejecutable con los resultados obtenidos.
- **Despliegue:** Es la última etapa donde interviene el servidor de automatización. En ésta el ejecutable es desplegado en el servidor productivo para finalmente llegar a los usuarios.
- **Operación:** Etapa de monitoreo y pruebas de usuario donde el sistema se encuentra en operación.

Para este trabajo de grado solo se desarrollarán temas sobre las etapas de planificación, código y pruebas. Dichas etapas están identificadas con fondo rojo en la Figura 1.

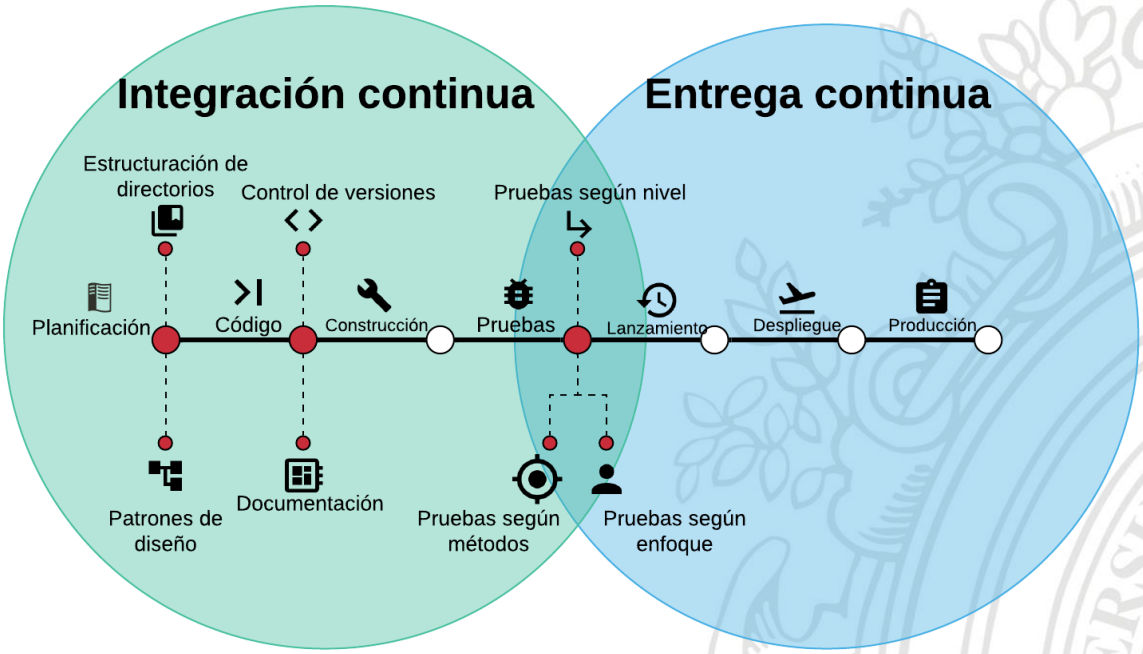

Figura 1. Etapas de integración y entrega continuas

1.4.2.1. Etapa de planificación

La etapa de planificación puede ser tan extensa y compleja como el equipo a cargo del proyecto lo requiera. En esta etapa se tratan temas como: organización del equipo, verificación de los presupuestos, identificación de requisitos funcionales, estructuración de directorios y definiciones acerca del software. Los dos últimos temas son los que se abordaran en esta sección de la tesis.

1.4.2.1.1. Patrones de diseño

Este concepto hace referencia a implementaciones de código comunes usadas para solucionar problemas de algoritmia. Un patrón NO ES un algoritmo, más bien se identifica con una abstracción de código. Es decir, un algoritmo se entiende como una secuencia de pasos lógicos para brindar una solución específica a un determinado problema. Por otro lado, un patrón es una visualización de alto nivel donde se proponen estrategias para dar soluciones generales a problemas específicos. Debido a esto, un patrón puede tener diferentes implementaciones en un mismo lenguaje sin perder coherencia [18].

1.4.2.1.1.1. ¿Son necesarios los patrones de diseño?

Los patrones de diseño representan soluciones comunes a problemas de programación orientada a objetos (POO). En ocasiones los desarrolladores de manera inconsciente implementan dichas abstracciones para solucionar problemas cotidianos de POO. Es decir, podemos estar usando patrones sin saberlo. Sin embargo, su uso y aprendizaje supone un aumento en la eficiencia al encontrar maneras rápidas, verificadas y optimizadas para solucionar un problema. En resumen, se puede desarrollar sin usar patrones de diseño, pero éstos representan un sello de calidad y estabilidad y el software. [18].

1.4.2.1.1.2. División de patrones de diseño

Los patrones se clasifican según su nivel de detalle, complejidad y nivel de implementación en una aplicación. Según las necesidades específicas de un sistema los patrones se dividen en:

- **Patrones de creación:** Permiten crear de manera eficiente elementos en nuestro sistema. Promueven la reusabilidad y control de errores de ejecución.
- **Patrones estructurales:** Explican cómo se deberían unir grandes estructuras de datos y como se relacionan entre sí.
- **Patrones de comportamiento:** Se preocupan por la comunicación eficiente entre elementos y delegan responsabilidades en los objetos que intervienen.

1.4.2.1.1.3. Patrones de diseño de mayor difusión

- **Patrón constructor (builder pattern):** Permite construir objetos paso por paso, garantizando que el código mantenga su integridad y que el tipado se conserve. Hace parte de los patrones de creación y nace con la necesidad de evitar el desarrollo de múltiples constructores de gran tamaño. Facilita la mantenibilidad y las pruebas de una clase.

  La Figura 2 [18], nos permite identificar la estructura con la cual se diseña un código que sigue el patrón builder. En este patrón de diseño puede apreciarse la necesidad de múltiples implementaciones de la clase, mediante un objeto constructor que es manipulado por el objeto director.

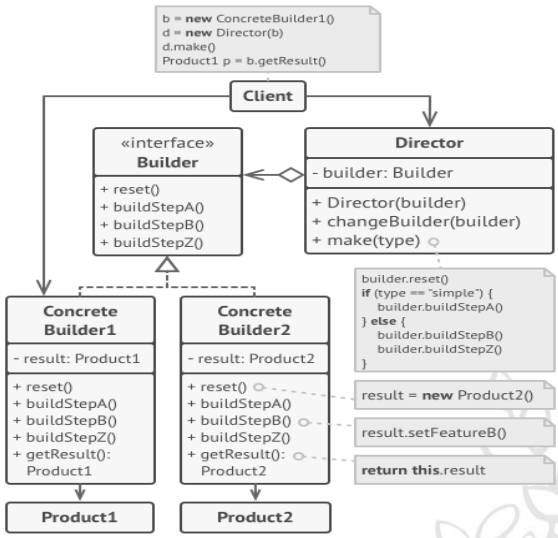

Figura 2. Estructura de patrón builder

- **Patrón fabrica abstracta (Abstract factory pattern):** Es un patrón creacional que permite producir familias de objetos relacionados sin especificar una clase en concreto. Nace con la necesidad de evitar construir nuevo código cuando categorías de objetos nuevos entran a la lógica del negocio. Las variantes del negocio pueden abstraerse y crear objetos genéricos que se adapten a características específicas.

  La Figura 3 [18], nos permite identificar la estructura con la cual se diseña un código que sigue el abstract factory pattern. En la imagen puede apreciarse como los objetos se definen de diferentes factorías permitiendo que conserven la herencia del tipo principal. Sin embargo, son definidos con características distintas según de la factoría que provengan. Finalmente, el cliente consume la factoría que necesite según el objeto que desee crear.

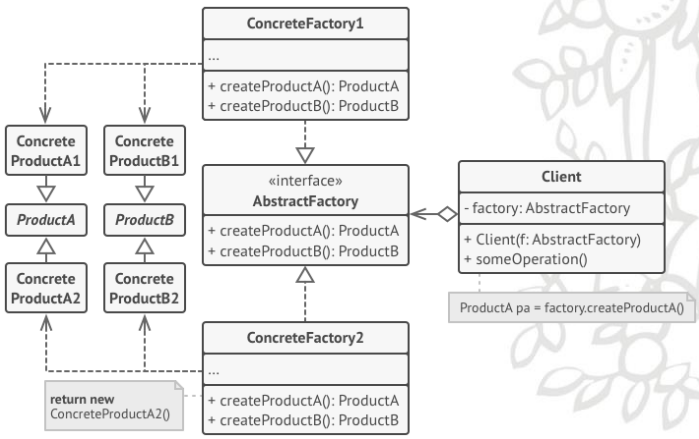

Figura 3. Estructura de patrón abstract factory

- **Patrón uni-instancia (singleton pattern):** Patrón creacional que garantiza que una clase solo tenga una instancia en todo el proceso de ejecución del código. Es decir, garantiza un único acceso global a la instancia en cuestión. Este patrón nace del problema generado por información duplicada en diferentes instancias en el código. También garantiza el acceso optimo a recursos compartidos, permitiendo que los datos mantengan su integridad.

La Figura 4 [18], nos permite identificar la estructura con la cual se diseña un código que sigue el singleton pattern. En la imagen puede apreciarse como el cliente solo tiene acceso a una instancia de dicha clase y mediante ésta se gestionan todos los cambios.

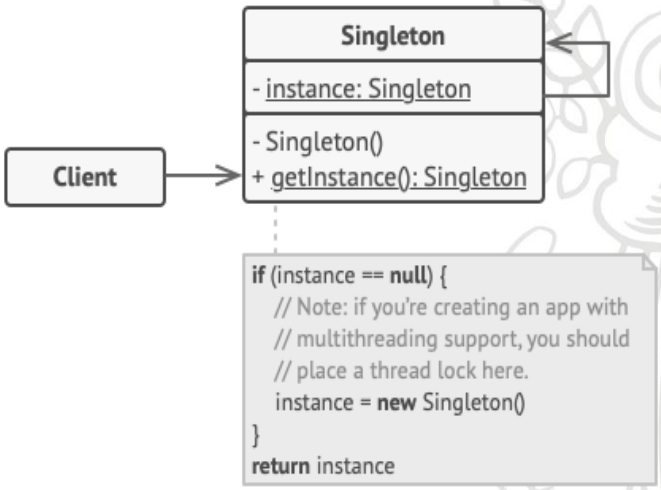

Figura 4. Estructura de patrón singleton

1.4.2.1.2. Estructuración de código

La etapa de estructuración de código puede resultar confusa al lector, dado que estructurar código no implica desarrollarlo. Por el contrario, implica un proceso organización donde se define cual es la estructura idónea que deben tener los archivos y directorios de un proyecto para garantizar orden, claridad y flujos de trabajo agiles. Normalmente es un tema muy poco discutido y tiende a realizarse de manera empírica. Cabe resaltar que algunos desarrolladores consideran que la estructuración de código hace parte de la etapa de código. Sin embargo, como estructurar código no implica desarrollar código en esta tesis se considera como parte de la etapa de planificación.

1.4.2.2. Etapa de código

Los elementos que integran esta etapa son bastante discutidos y en pocas ocasiones existe concertación sobre los temas que agrupa. Sin embargo, existen dos grandes visiones alrededor

de esta discusión. La primera considera que la etapa de código únicamente incluye el proceso de creación del código junto con el control de versiones. Mientras, la segunda visión considera el proceso de creación de código como conjunto integral de prácticas que agrupan: la creación, documentación y control de versiones del código. Para este trabajo de grado tendremos como referencia la segunda visión de la etapa de código.

1.4.2.2.1. Documentación.

El proceso de documentar busca garantizar que el código sea legible, de fácil comprensión y altamente mantenible para cualquier desarrollador, ya sea un miembro del equipo de trabajo o un futuro desarrollador que tenga la tarea de brindar soporte al código. Actualmente, en la industria existen dos tipos de vertientes en el desarrollo del código. Los desarrolladores que piensan que el código en sí mismo debería ser suficiente para brindar comprensión de su propósito, y los desarrolladores que piensan que un buen código debe estar acompañado de una excelente documentación.

1.4.2.2.1.1. ¿Por qué documentar es tan importante?

"Code is more often read than written"

— Guido Van Rosssum

El desarrollo de código tiene principalmente dos audiencias: los desarrolladores que tendrán que modificar o extender las funcionalidades del código y los usuarios que harán uso estas funcionalidades en sus aplicaciones. Las dos audiencias juegan un rol importante, una permite que el proyecto alcance nuevos horizontes y la otra permite que el proyecto tenga un impacto real.

Para los desarrolladores que usan un proyecto ya planteado son fundamentales los comentarios dentro del código, las variables autodescriptivas, el orden en el sistema de directorios, entre otros detalles de arquitectura. Por otro lado, para los usuarios es fundamental la documentación escrita, ejemplos de uso, guías de las APIs (Application Programming Interface), entre otros documentos que brindar valor agregado. En resumen, no importa cuán bueno sea el software desarrollado si nuestra documentación no está a la altura, dado que nadie deseará enfrascarse en un proyecto sin soporte.

1.4.2.2.1.2. ¿Por qué necesito documentar el código?

Documentamos el código porque deseamos que pueda ser mantenible o para dar soporte a quien esté trabajando con él [19]. Algunas razones específicas para documentar el código son:

- Es bueno para la transferencia de conocimiento. No todo el código es igualmente obvio, éste puede tener algunos algoritmos complejos o soluciones personalizadas que no son lo suficientemente claras para otros desarrolladores.
- Ayuda a solucionar problemas de producción. Si existe alguna dificultad con la aplicación contar con la documentación adecuada puede acelerar el tiempo para

solucionar los problemas. Conocer los detalles del producto y los detalles de la arquitectura es una tarea que lleva mucho tiempo, por ende, la documentación clara y eficiente de los puntos críticos del código permite encontrar fallas fácilmente.
- Ayuda a administrar mejor las integraciones y módulos adicionales.

1.4.2.2.1.3. ¿Cómo documentar correctamente?

No tener documentación es tan malo como tener documentación excesiva o inapropiada. Algunas reglas básicas para crear documentación de código útil y manejable son:

- **Documentación simple y concisa:** Se debe seguir el principio de no repetición. No es necesario comentar cada línea del código, se deben utilizar los comentarios para explicar algo que no es evidente.
- **Documentación actualizada:** Es mejor documentar el código paso a paso, tal como está escrito, en lugar de comentar cuando el proyecto finalice. Al comentar al instante se ahorra tiempo, y la documentación será más precisa y completa.
- **Mantener los cambios en desuso**: Documentar nuevas características o complementos es bastante obvio. Sin embargo, también debe documentar las características en desuso, capturando cualquier cambio en el producto.
- **Mantener un lenguaje simple y con formato adecuado:** Los documentos de código generalmente se escriben en inglés para que cualquier desarrollador pueda leer los comentarios, independientemente de su idioma nativo. Las mejores prácticas para la redacción de documentación requieren de lenguaje imperativo, tiempo presente, preferiblemente la voz activa y en tercera persona.
- **Combine herramientas de documentación automatizadas y aportes humanos**: La automatización acelerará el proceso, pero una persona puede hacer que la documentación del código sea comprensible al tiempo que agrega un toque más personal.

1.4.2.2.1.4. Documentar VS Comentar.

"Code tells you how; Comments tell you why."

— Jeff Atwood (aka Coding Horror)

En general, comentar es describir o explicar el código para que otros desarrolladores puedan seguir el flujo de éste. El público principal son los mantenedores y desarrolladores del código. Junto con un código bien escrito, los comentarios ayudan a guiar al lector a comprender mejor el código, su propósito y diseño. Los comentarios hacen parte de la documentación, pero un código bien comentado, no implica que este bien documentado.

Documentación, como su nombre lo indica, es un conjunto de documentos que buscan proveer información. Particularmente, en los ambientes técnicos, se puede entender como una guía de referencia que indica el funcionamiento, operatividad y uso del código. Si bien puede ser útil para los desarrolladores, el público objetivo son los usuarios.

La siguiente sección describe cómo y cuándo comentar el código.

1.4.2.2.1.5. Momentos útiles donde se usan los comentarios

- **Planificación y revisión:** Cuando se desarrollan nuevas porciones de código, puede ser apropiado usar primero los comentarios como una forma de planificar o delimitar una sección. Estos comentarios deben eliminarse al finalizar el proyecto.
- **Descripción del código**: Se usan para explicar una sección específica del código.
- **Descripción algorítmica:** Cuando se utilizan algoritmos, especialmente complicados, puede ser útil explicar cómo funciona el algoritmo o cómo se implementa dentro del código. También puede ser apropiado describir por qué se seleccionó un algoritmo específico sobre otro.
- **Etiquetado:** Se utiliza para etiquetar secciones específicas de código donde se encuentran problemas conocidos o áreas de mejora. Algunos ejemplos son: BUG (Error), FIXME (Arréglame) y TODO (Por hacer).

1.4.2.2.1.6. Datos útiles para comentar

Los comentarios al código deben ser breves y enfocados. Se debe evitar usar comentarios largos cuando sea posible. Además, se debe usar las siguientes cuatro reglas esenciales según lo sugerido por Jeff Atwood [20]:

- Mantener los comentarios lo más cerca posible del código que se describe. Los comentarios que no están cerca de su código descriptivo son frustrantes para el lector y se pierden fácilmente cuando se realizan actualizaciones.
- Se debe evitar utilizar formatos complejos (como tablas o figuras ASCII). Los formatos complejos conducen a contenido que distrae y puede ser difícil de mantener con el tiempo.
- Se debe evitar incluir información redundante. Suponga que el lector del código tiene una comprensión básica de los principios de programación y la sintaxis del lenguaje.
- Diseñar el código para comentarlo. La forma más fácil de entender el código es leerlo. El código debe diseñarse utilizando conceptos claros y fáciles de entender, así, el lector podrá conceptualizar rápidamente la intención.

Vale la pena resaltar que los comentarios están diseñados para el lector, incluido usted. En resumen, los comentarios ayudan a comprender el propósito y el diseño del software.

El proceso de comentar un código depende en gran medida del lenguaje de programación elegido, según el lenguaje existen convenciones o normas que delimitan la manera de comentar e incluso la estructura del comentario. La mayoría de los lenguajes de programación cuentan con herramientas que automatizan el proceso de comentar y organizar dicha documentación. En este documento vamos a tocar docstring para Python y JavaDocs para Java.

1.4.2.2.2. Sistema de control de versiones (VCS, Version Control System)

El control de versiones es un sistema que registra los cambios en un archivo o conjunto de archivos a lo largo del tiempo para que se pueda recuperar versiones específicas más adelante. Usar un VCS también implica que, si se arruinan o pierden los archivos, será posible recuperarlos fácilmente [3].

Los sistemas de versiones se clasifican en tres categorías: locales, centralizamos y distribuidos (ver Figura 5)

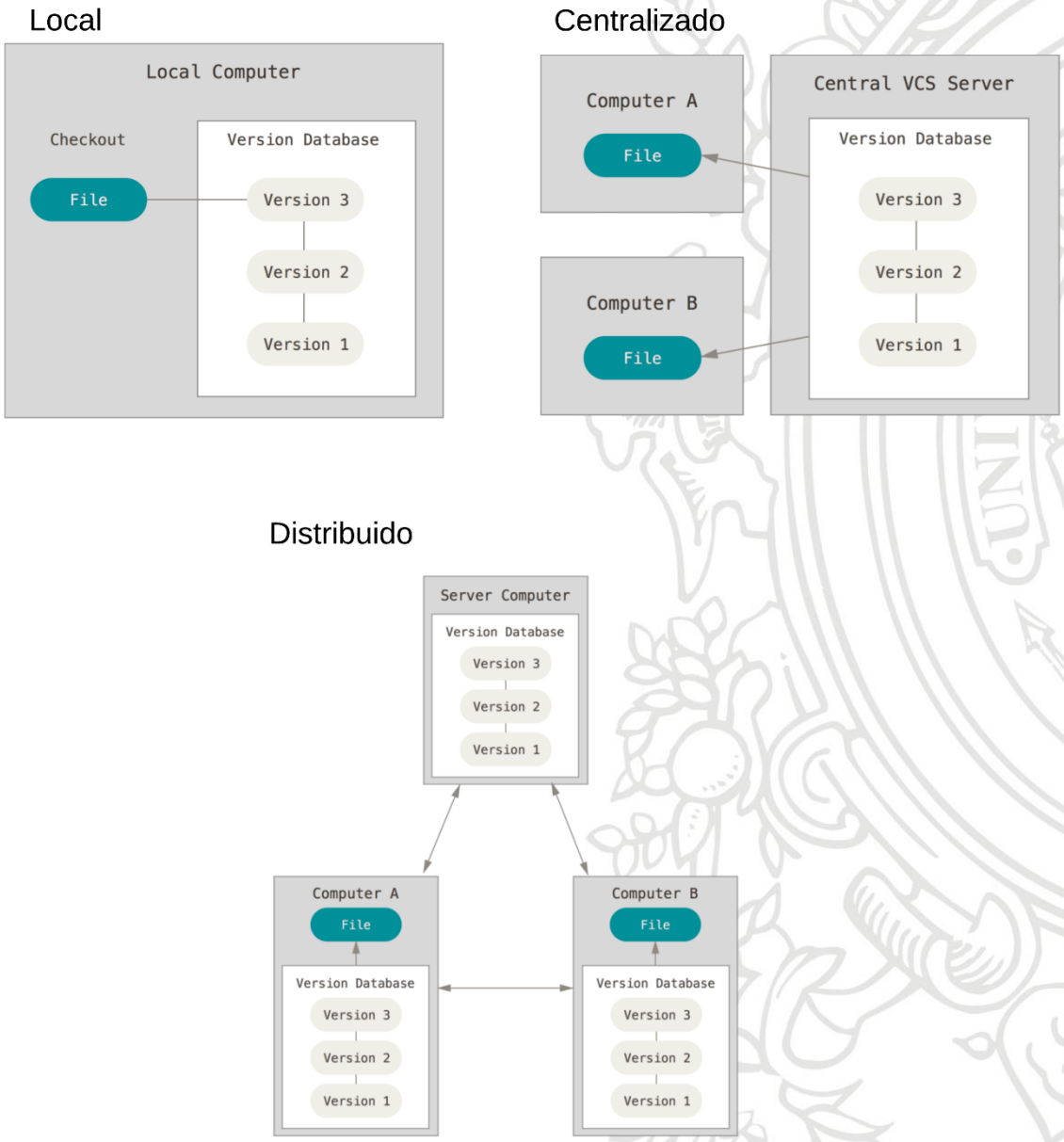

Figura 5. Tipos de sistemas de control de versiones

1.4.2.2.2.1. Sistemas de Control de Versiones Locales

Un método de control de versiones, usado por muchas personas, es copiar los archivos de un directorio a otro (quizás indicando la fecha y hora en que lo hicieron, si son ingeniosos). Este método es muy común porque es sencillo, pero también es tremendamente propenso a errores. Es fácil olvidar en qué directorio te encuentras y guardar accidentalmente en el archivo equivocado o sobrescribir archivos que no querías. Para afrontar este problema, los programadores desarrollaron los VCS locales, los cuales contienen una simple base de datos en la que se lleva el registro de todos los cambios realizados a los archivos. Una de las herramientas de control de versiones más popular fue un sistema llamado RCS (Revision Control System). Incluso, todavía el famoso sistema operativo Mac OS X incluye el comando rcs cuando instalas las herramientas de desarrollo. RCS funciona guardando conjuntos de parches, es decir, las diferencias entre archivos en un formato especial en disco. Además RCS es capaz de recrear cómo era un archivo en cualquier momento a partir de dichos parches [21].

1.4.2.2.2.2. Sistemas de Control de Versiones Centralizados

Los VCS Centralizados (CVCS, Centralized Version Control System) fueron desarrollados para solucionar las dificultades de comunicación entre los desarrolladores de un mismo proyecto. Herramientas como CVS, Subversion y Perforce, tienen un único servidor que contiene todos los archivos versionados y varios clientes que descargan los archivos desde el servidor central. Este ha sido el estándar para el control de versiones por muchos años.

Los CVCS ofrecen muchas ventajas, especialmente frente a VCS locales. Por ejemplo, todas las personas saben hasta cierto punto en qué están trabajando los otros colaboradores del proyecto. Los administradores tienen control detallado sobre qué puede hacer cada usuario, y es mucho más fácil administrar un CVCS que tener que lidiar con bases de datos locales en cada cliente. Sin embargo, esta configuración también tiene serias desventajas, la más obvia es el punto único de fallo que representa el servidor centralizado. Si el servidor se cae durante una hora, entonces durante esa hora nadie podrá colaborar o guardar cambios en archivos en los que hayan estado trabajando. Si el disco duro en el que se encuentra la base de datos central se corrompe, y no se han realizado copias de seguridad adecuadamente, se perderá toda la información del proyecto, con excepción de las copias instantáneas que las personas tengan en sus máquinas locales. Los VCS locales sufren de este mismo problema, cuando tienes toda la historia del proyecto en un mismo lugar, te arriesgas a perderlo todo [21].

1.4.2.2.2.3. Sistemas de Control de Versiones Distribuidos

Los sistemas de Control de Versiones Distribuidos (DVCS, Distributed Version Control System), ofrecen soluciones para los problemas que presentan los CVCS. En un DVCS (como Git, Mercurial, Bazaar o Darcs), los clientes no descargan una copia instantánea de los archivos, sino que descargan el repositorio en su totalidad. De esta manera, si un servidor deja de

funcionar y estos sistemas estaban colaborando a través de él, cualquiera de los repositorios disponibles en los clientes puede ser copiado al servidor con el fin de restaurarlo. Cada clon es realmente una copia completa de todos los datos [21].

1.4.2.3. Etapa de pruebas

La etapa de pruebas es el punto del flujo de CI/CD donde se verifica que el código desarrollado cumpla con los lineamientos definidos. En esta etapa se unen la integración y entrega continúa creando un punto hibrido que implica desarrollo de código y automatización.

1.4.2.3.1. ¿Por qué son importantes las pruebas en el software?

En la actualidad, gran parte de nuestra cotidianidad depende de software y de sistemas de información. Confiar en ellos es trascendental para el funcionamiento de la sociedad. Pero ¿Qué pasa si estos sistemas resultan ser defectuosos? Un pequeño error en el sistema genera un gran impacto en las empresas en términos económicos. Por lo tanto, para entregar un producto de calidad, se necesitan pruebas de software en el proceso de desarrollo.

Las pruebas de software hacen referencia a un conjunto de procedimientos que permiten evaluar la funcionalidad de una aplicación, para así determinar si el software desarrollado cumple con los requisitos.

Realizar pruebas puede parecer un proceso tedioso. Sin embargo, no tiene por qué ser así, dado que su realización puede comenzar con pasos pequeños en los que el desarrollador tenga claridad del estado de pocos componentes. Luego, a medida que se requiera, se puede ir aumentando la cantidad de pruebas. No siempre un código debe tener un cubrimiento total en las pruebas. Lo importante es que brinden valor agregado al software y no que se conviertan en un vertedero de código innecesario.

1.4.2.3.2. Clasificación de las pruebas

Las pruebas en el software se clasifican según: los métodos usados para realizarlas, los enfoques con que se abordan y los niveles de detalle con los cuales se analizan.

1.4.2.3.2.1. Según el método

- **Métodos estáticos:** Buscan encontrar defectos sin ejecutar código, es decir, implican metodologías de análisis sobre los requisitos funcionales y modelos de arquitectura. Generalmente se usan en etapas tempranas del proyecto para identificar si las decisiones de diseño van por buen camino.
- **Métodos dinámicos:** Implican pruebas en tiempo de ejecución. Su objetivo es asegurar que todas las piezas encajen y que la ejecución del código se encuentre conforme a las especificaciones de diseño proyectadas.

1.4.2.3.2.2. Según el enfoque

- **Caja blanca:** La caja blanca también llamada caja de cristal, es un proceso de software que se basa en la estructura del código interno de las aplicaciones. En las pruebas de caja blanca, se utiliza una perspectiva interna del sistema, así como habilidades de programación, para diseñar casos de prueba. Esta prueba generalmente se realiza a nivel de unidad, es decir, se evalúa una pieza de código especifica (función, método de clase o script).
- **Caja negra:** También llamada Prueba de comportamiento / Prueba basada en especificaciones / Pruebas de entrada-salida. La caja negra es un método de prueba de software en el que los evaluadores validan la funcionalidad del software sin mirar la estructura interna del código.

1.4.2.3.2.3. Según el nivel de detalle

- **Pruebas unitarias:** Son las pruebas con el nivel de detalle más preciso. Implican probar un componente individual separado del resto de código. Estas pruebas se usan para conocer la trazabilidad del funcionamiento de un componente sin importar su comunicación el resto de los elementos del programa.
- **Pruebas de integración:** Algunas veces llamadas pruebas de compatibilidad. Las pruebas de integración implican unir fragmentos de código para valorar su comportamiento ante la integración. Como su nombre lo indica, éstas buscan verificar la comunicación entre dos o más puntos de la aplicación, suponiendo que cada uno funciona correctamente.
- **Pruebas de sistema:** Son pruebas que permiten verificar el funcionamiento de la aplicación. Éstas implican usar un alto nivel de abstracción donde se entienda todo el conjunto como un sistema. Para verificar las salidas deseadas es necesario verificar cada entrada, además de probar la experiencia de usuario en la aplicación.

1.4.3. Automatización de software

El proceso de automatizar código es más simple de los que podríamos llegar a pensar. Los proyectos de software son automatizados en mayor o menor nivel con el propósito de adaptarse a las necesidades del mercado y ofrecer mayor calidad en el producto. Un software se considera automatizado cuando cumple con los siguientes requisitos:

- Verificación constante del estado del código, es decir, identificación de malas prácticas y verificación de porcentaje de cubrimiento de las pruebas.
- Implementación de un sistema de control de versiones integrado con un sistema de despliegue a pruebas y producción.
- Implementación de sistema de compilación y empaquetado para el lanzamiento automático de versiones.
- Verificación de pruebas automáticas antes de la etapa de despliegue.

- Proceso de monitoreo automático entre etapas de CI/CD sin interacción humana.

Desarrollar dichas tareas de manera independiente es un trabajo titánico que requiere la sincronizar multitud de procesos en un solo punto. Para poder llevar a cabo dicha sincronización de manera eficiente se diseñaron los llamados "servidores de automatización".

1.4.3.1. Servidores de automatización

Según Microsoft [22], un servidor de automatización es una aplicación que expone objetos programables (objetos de automatización) a otras aplicaciones (clientes de automatización). La exposición de objetos programables permite a los clientes automatizar ciertos procedimientos al acceder directamente a los objetos y la funcionalidad que el servidor pone a disposición.

En palabras simples un servidor de automatización busca resolver la necesidad de los despliegues de aplicaciones a ambientes productivos o de usuario final. Este se encarga de tomar el código, empaquetarlo y ejecutarlo en el ambiente productivo. Entre los servidores de automatización más conocidos están: Jenkins, Circle CI, Travis, appVeyor y CodeShip.

1.4.3.1.1. ¿De qué se encarga un servidor de automatización?

Los servidores de automatización sirven como punto de integración entre diferentes etapas del proceso de CI/CD. Cabe resaltar que cada software que sirve como "servidor de automatización" implementa funcionalidades diferentes. Las funcionalidades más conocidas de los servidores de automatización son:

- Integración automática con los repositorios de control de versiones para ejecutar procesos mediante acciones que "disparan" estados. Es decir, permiten ejecutar procesos cuando en el repositorio se sube, modifica o elimina código.
- Descarga y control de versiones para las dependencias.
- Modificación del código para inyectar variables privadas como: tokens, contraseñas y claves de acceso.
- Empaquetado de código para generar lanzamientos automáticos.
- Gestión vía comandos bash para despliegue de aplicaciones en ambientes productivos o de pruebas.
- Gestión de perfiles y ambientes de desarrollo.
- Monitoreo del proceso con sistema de alertas.
- Gestión de roles para control de acceso a las etapas de CI/CD.
- Configuraciones rápidas para los lenguajes de programación más usados.

1.4.3.1.2. Beneficios de un servidor de automatización

Los beneficios de un servidor de automatización pueden resumirse en tres conceptos: velocidad, mantenibilidad y precisión.

1.4.3.1.2.1. Velocidad

La velocidad en el desarrollo de software puede manifestarse de las siguientes maneras:

- Retroalimentación de los cambios realizados.
- Construcción, verificación y ejecución centralizados.
- Respuesta eficaz a los cambios que demanda el aplicativo.
- Configuración y curva de aprendizaje.

1.4.3.1.2.2. Mantenibilidad

Los servidores de automatización garantizan estabilidad al proceso desarrollo, eliminando la posibilidad de fallas intermitentes causadas por errores de gestión de los ambientes productivos. Operar y escalar la infraestructura requiere alta disponibilidad, mínima tolerancia a los errores y constante mantenimiento que garantice condiciones óptimas de funcionamiento. Estás características difícilmente pueden ser suplidas por una persona o un grupo, de aquí que los servidores de automatización sean la opción por elegir en el proceso de desarrollo.

1.4.3.1.2.3. Precisión

El beneficio más relevante de los servidores de automatización es su capacidad de ejecutar acciones de manera precisa. Los servidores de automatización garantizan que el proceso de construcción, verificación y despliegue sea exactamente ejecutado como fue definido. Esta característica es especialmente relevante cuando se tiene que gestionar múltiples aplicaciones con características completamente diferentes.

En esta sección solo hablaremos de Jenkins pues el servidor de automatización más usado en el mundo. Además, de ser la única alternativa OpenSource del mercado. En la Figura 6 se puede visualizar las ventajas de Jenkins frente a la competencia [23].

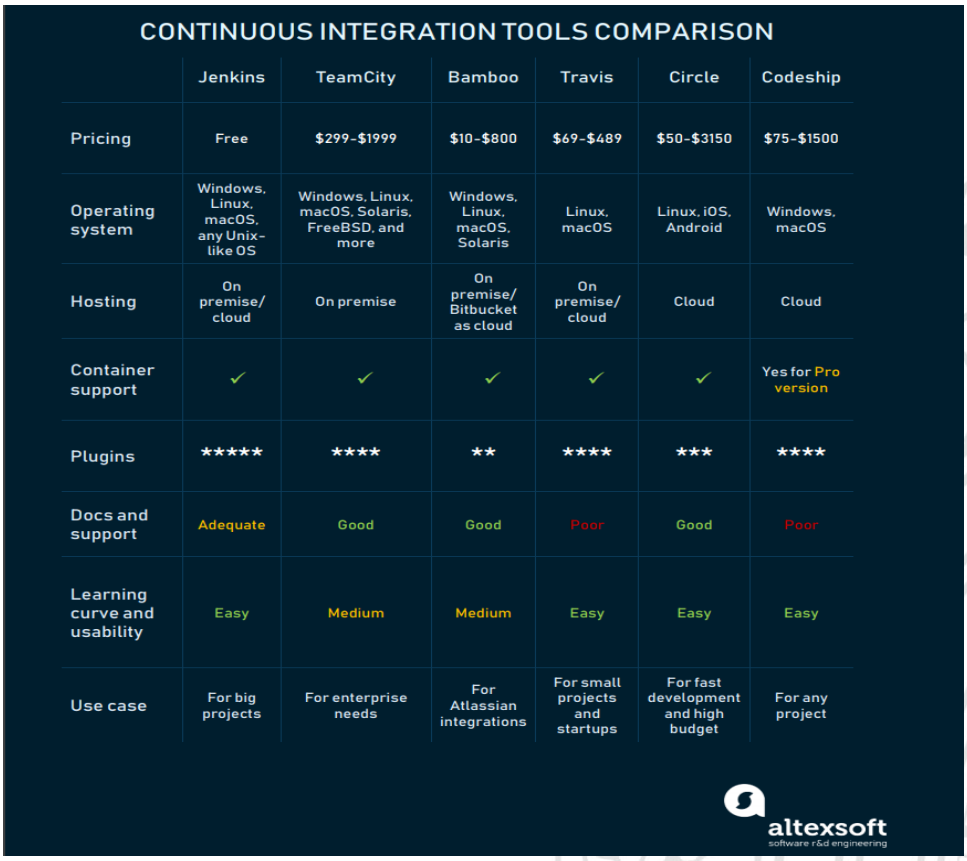

Figura 6. Comparativa de servidores de automatización

1.4.3.2. Jenkins

Jenkins es un software de automatización que ofrece una manera simple de configurar procesos de integración y entrega continua para casi cualquier combinación de lenguajes de programación. Hoy Jenkins es el servidor de automatización de código abierto líder con unos 1.400 complementos para admitir la automatización de todo tipo de tareas de desarrollo. Jenkins originalmente estaba tratando de resolver, la integración y la entrega continua de código Java mediante la construcción de proyectos, ejecución de pruebas y el análisis de código estático. Sin embargo, estos son solo algunos de los procesos que las personas automatizan con Jenkins. Esos 1,400 complementos abarcan cinco áreas: plataformas, interfaz de usuario, administración, administración de código fuente y cada día con mayor frecuencia, administración de compilación [24].

2. Lineamientos para promover la calidad en el desarrollo del software de los proyectos del grupo GITA

2.1. Metodología

Con el propósito de establecer un conjunto de buenas prácticas para mejorar la calidad del software al interior del grupo GITA se realizaron diferentes actividades:

2.1.1. Reconocimiento de los proyectos en desarrollo

En esta etapa se realizaron encuentros con los docentes, y estudiantes de maestría encargados, de las diferentes líneas del grupo GITA (Modelamiento, Procesamiento de Señales y Óptica). El objetivo de dichos encuentros era identificar:

1. ¿Cuáles eran los proyectos en ejecución al interior del grupo o en sus cursos a cargo?
2. ¿Qué lenguajes de programación eran utilizados?
3. ¿Cuáles eran las estrategias utilizadas para su desarrollo?
4. ¿Qué metodologías usaban para mantener su código actualizado?
5. ¿Cómo respaldaban sus archivos?
6. ¿Cómo documentaban su proyecto?
7. ¿Qué normas seguían para probar el código?
8. ¿Cómo versionaban su código?

2.1.2. Identificación del estado de los proyectos

De la etapa anterior se identificó que los lenguajes que predominaban en el desarrollo de proyectos era Python y Java. De aquí que este trabajo de grado esté orientado en dichos lenguajes.

De d

También se identificó que no se implementaba ningún tipo de metodología para el desarrollo de software como tal y que, por lo tanto, no existía un conjunto de buenas prácticas que permitieran el manejo de estándares de calidad. Los hallazgos más relevantes encontrados en el código fueron:

- La documentación era muy limitada, se concentraba en comentarios que realizaba el autor para sí mismo y no en la descripción del código.

- Los algoritmos no se describían, dejando dudas sobre su funcionamiento e implementación.
- Los pasos para ejecutar un proyecto no se encontraban correctamente estructurados y en algunos casos no existían.
- Las clases y funciones eran poco descriptivas y en algunos casos su definición generaba confusión.

2.1.3. Recopilación de estándares de calidad

En esta etapa se realizó una revisión de los principales estándares de calidad para el desarrollo de software, de acuerdo con los hallazgos identificados en la sección 2.1.2.

En esta etapa fue necesario realizar nuevos encuentros con los miembros del grupo de investigación, para depurar el compendio de estándares de calidad recopilados. Este proceso de depuración obedeció a que, dadas las áreas de investigación de cada línea, se debía identificar cuáles estándares brindaban mayores beneficios de acuerdo con el campo de aplicación.

2.1.4. Definición de estrategias

En esta etapa del proceso la información objetivo se encontraba clara. De aquí que se procedió a la definición de lineamientos que permitieran mejorar la calidad del software. Aquí se construyeron un conjunto de guías para:

- Implementar sistemas de control de versiones dentro de las dinámicas del grupo de investigación.
- Documentar el código en los lenguajes más usados dentro del grupo de investigación (Python y Java). Es de resaltar que las guías de documentación son lo suficientemente genéricas para poder extrapolar las definiciones de estos dos lenguajes a otros.
- Crear compilado con la explicación y uso de las pruebas en los proyectos de desarrollo de software.

2.1.5. Desarrollo de mecanismos de socialización

En esta etapa, con el propósito de consolidar toda la información desarrollada en un solo lugar para facilitar su consulta, se desarrolló una Wiki. Esta wiki se alojó en el servidor de ingeniería de GitLab, de tal manera que estuviera accesible solo para los miembros de grupo de investigación GITA. También se almacenaron las guías para los lineamientos del proyecto en dicho servidor. Y finalmente resultó una unidad compartida que recoge toda la información del proyecto de grado, junto con las cuatro sesiones de socialización (video de la socialización y diapositivas).

2.1.6. Implementación de proyecto piloto

El proyecto piloto condensó todos los objetivos del trabajo de grado en un solo proyecto de software. Para éste, se exploraron los resultados obtenidos en los lineamientos de software y se generó un proyecto que permitiera servir de guía a los integrantes del grupo de investigación GITA.

Las primeras etapas del proyecto piloto consistieron en reuniones de realimentación con los miembros del grupo de investigación. También se realizaron reuniones con monitores de cursos avanzados de ingeniería de telecomunicaciones y docentes. Finalmente, de las reuniones resultaron 4 principios esenciales que deberían cumplirse en el proyecto piloto para que tuviera un impacto real en el desarrollo de software interno del grupo.

1. El proyecto piloto debía incluir todos los aspectos de la metodología.
2. La ejecución inicial de la prueba debía ser sencilla y fácil de replicar por otros usuarios.
3. Se debía partir desde un código simple y fácil de entender para facilitar el aprendizaje de la metodología para los miembros menos adaptados a la programación.
4. El proyecto piloto debía tener una jornada de socialización que permitiera a los miembros actuales del grupo replicar los conocimientos aprendidos.

Teniendo en cuenta los principios definidos en las reuniones, se propuso utilizar el proyecto beermeeting del curso de servicios telemáticos como proyecto piloto. De tal manera, que los propios estudiantes del curso identificaran los beneficios y complejidad de la metodología.

2.2. Elección de patrones de diseño

Elegir un patrón de diseño para solucionar de manera de eficiente un problema se ha convertido en uno de los mayores retos en la programación orientada a objetos (POO). Los patrones permiten visualizar soluciones elegantes y de alto rendimiento ante ciertos escenarios que comúnmente se presentan a los desarrolladores de código. Según las necesidades requeridas en el ambiente de trabajo se puede determinar qué patrón de diseño usar. Es decir, un lenguaje como Python a pesar de usarse para hacer múltiples trabajos, tiene patrones más útiles según su aplicación.

Cuando se desarrolla una aplicación y se decide comenzar con el paradigma de programación POO, es necesario visualizar: cómo se crearán de los objetos, la manera en que dichos objetos se comunicarán con otros y la estructura de los objetos como tal. En definitiva, es necesario pensar en cómo cada pieza del código interactuará con el mismo y otros códigos.

En el grupo de investigación GITA, se identificó que la mayoría de los códigos, vinculados a los proyectos en desarrollo, pueden adaptarse a los siguientes patrones de diseño:

- **Singleton pattern / patrón de una sola instancia:** Este patrón de diseño centraliza el uso de instancias, permitiendo que el acceso a recursos importantes no sea desde múltiples lugares. Es decir, el acceso a estos recursos es compartido de manera eficiente y segura. El patrón uni-instancia se utiliza principalmente en el acceso a bases de datos, archivos de utilidades o grupos de información.
  En GITA, gran cantidad de proyectos necesitan del procesamiento de datos, lo que implica un uso extensivo de las bases de datos. Por esto, se pudo identificar al patrón singleton como uno de los patrones indispensables para el desarrollo de código al interior del grupo.

- **Builder pattern / patrón constructor:** Como su nombre lo indica el builder pattern tiene como principal objetivo ensamblar objetos. Este patrón se usa principalmente en las aplicaciones que necesitan objetos con múltiples constructores y de atributos similares. En el grupo de investigación GITA gran parte de los proyectos necesitaban definir objetos de manera única. Esta necesidad llevó a que se considerará el builder pattern como una alternativa que permitiera definir fácilmente los objetos e identificar sus propiedades en cualquier parte del código. Este patrón también llamó la atención por su facilidad para permitir comparaciones entre objetos.

- **Abstract factory pattern / patrón de fabrica abstracta:** Su nombre, aunque algo desconcertante es ilustrativo sobre la finalidad de este patrón. El patrón fabrica abstracta permite crear objetos con características similares controlando la herencia y las características finales del objeto. Este patrón se usa principalmente en las aplicaciones que incluyen selección de objetos como inventarios, sistemas de gestión de presupuestos y tiendas. Este patrón resulta de especial utilidad para el grupo de investigación pues permite categorizar objetos con mayor facilidad. Además, cada una de las líneas manipula grandes cantidades de información que necesita ser clasificada y procesada por lo que este patrón resulta de gran utilidad.

Con el propósito de permitir a los desarrolladores del grupo identificar cuál es el patrón de diseño que más se ajusta a su desarrollo, se diseñó el diagrama de flujo ilustrado en la Figura 7. Este diagrama, mediante una serie de preguntas, facilitará la selección de un patrón que permite solucionar una dificultad en particular. Es importante mencionar que, las preguntas utilizadas en este diagrama están basadas fueron previamente socializadas con integrantes del grupo para validar la coherencia y funcionalidad del diagrama.

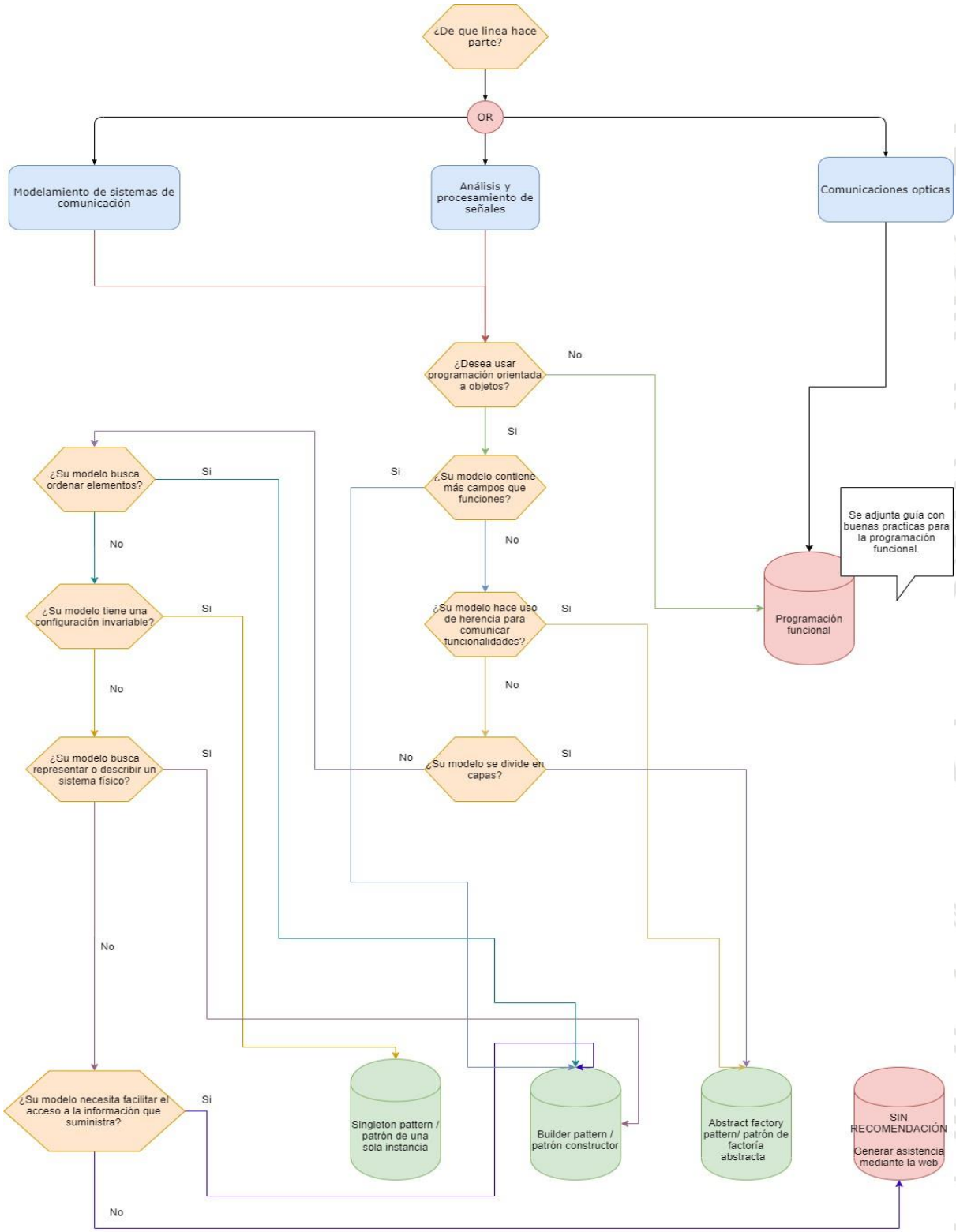

Figura 7. Diagrama para la selección de patrones de diseño al interior del grupo

2.3. Estructuración de código

"Estructurar" un proyecto se refiere al hecho de tomar decisiones para identificar la manera más eficiente en que éste cumple su objetivo. También implica identificar las ventajas que brinda el lenguaje de programación elegido y sacar provecho de sus puntos fuertes. En términos prácticos, estructurar un proyecto significa implementar código limpio y reusable, tal que su lógica y dependencias sean claras, al igual que los archivos y directorios que conforman el sistema.

La estructura del proyecto hace parte de la arquitectura de éste. Un proyecto necesita de una robusta estructura de archivos tanto como de un buen diseño de API y un buen estilo de programación. En definitiva, todo contribuye para mantener un ciclo de desarrollo saludable. Elegir una estructura débil puede causar grandes dificultades conforme el proyecto crece o es usado por otros desarrolladores. Los signos más comunes para identificar una estructura incorrecta son:

- **Romper el principio de única responsabilidad (Acoplamiento oculto):** Un cambio en una funcionalidad puede romper otras partes del código y hacer imposible de seguir las pruebas.
- **Dependencias circulares múltiples y desordenadas:** Al incluir dependencias en múltiples lugares del código, el sistema puede duplicar la carga de dichas dependencias causando un consumo de memoria excesivo. Además, identificar las dependencias que usa el proyecto puede ser una tarea complicada.
- **Uso intensivo del estado o contexto global:** El uso constante de variables globales indica que el código no define correctamente los entornos de propagación de sus variables. Ésto implica que en proyectos grandes el sistema tienda a desbordar la memoria.
- **Código espagueti:** Varias páginas de cláusulas if anidadas y bucles con una gran cantidad de código de procedimiento pegado y sin la segmentación adecuada. Por ejemplo, la sangría significativa de Python (una de sus características más controvertidas) hace que sea muy difícil mantener este tipo de código.
- **Código de raviolis:** Consta de cientos de pequeñas piezas lógicas similares, a menudo clases u objetos, sin la estructura adecuada.

En resumen, para estructurar un código permitiendo que éste sea reusable y mantenible es necesario: la definición de los archivos esenciales para el proyecto y una buena estructura de directorios.

2.3.1. Archivos generales

Los archivos generales de un proyecto se definen desde los estándares de calidad que la propia comunidad impone, para poder reusar y mantener el código de manera remota. Estos archivos no dependen del lenguaje de programación. Por el contrario, dependen de las directrices necesarias para acoplarse con los estándares de documentación y las licencias que rigen el uso específico de un proyecto.

2.3.1.1. README.md

Tipo: Archivo

Propósito: Esencialmente, un archivo README es un archivo de texto único (.txt o .md) que actúa como documentación integral para un proyecto o directorio. Suele ser la pieza de documentación y la página de destino más visibles para la mayoría de los proyectos de código abierto. Incluso el nombre de un archivo README en mayúsculas está diseñado para captar la atención de su lector y garantizar que sea lo primero que lea.

La mayoría de los repositorios actuales tiene una funcionalidad integrada que busca visualizar en la página de inicio el texto que contiene el archivo README. Un archivo README debería incluir:

1. **Nombre del proyecto:** Sin importar que tan pequeño es el alcance del proyecto es necesario darle una identificación clara que permita localizarlo.
2. **Resumen o introducción:** Es útil escribir la utilidad del proyecto en unas cuantas líneas.
3. **Prerrequisitos:** Contiene las librerías, lenguajes o frameworks que necesita el proyecto para funcionar correctamente. Esta sección se encuentra inmediatamente después de la introducción para evitar confusión a la hora de implementar el proyecto.
4. **Pasos de instalación**: Describe de manera detallada los pasos necesarios para ejecutar un proyecto. En caso de que existan diferentes ambientes de ejecución es necesario describir estos pasos en cada ambiente.
5. **Pasos de uso:** Después de la instalación es necesario una sección que describa el uso del proyecto con ejemplos simples que puedan ejecutarse rápidamente. Para este paso es muy útil tener un conjunto de pruebas que permitan identificar el comportamiento del proyecto ante casos particulares.
6. **Como contribuir:** Los proyectos de código libre siempre necesitan de una pequeña sección que resuma la manera en que es posible contribuir al proyecto.
7. **Contribuyentes:** Lista con los contribuyentes del proyecto.
8. **FAQ (Frequently Asked Questions):** Sección de preguntas frecuentes.
9. **Información de contacto:** Contiene los datos de contacto.

10. **Licencia:** Es la sección final del proyecto, indica los parámetros con los cuales se puede usar el código. Es poco probable que startups o compañías que confían en el software de terceros usen el código a menos que se indique que es viable.

2.3.1.2. LICENSE

Tipo: Archivo

Propósito: Esta es una de las partes más importante del repositorio. Este archivo contiene el texto completo de la licencia y las reclamaciones de derechos de autor. Si no está seguro que licencia debe elegir, puede visitar el sitio web https://choosealicense.com/ para elegir la licencia OpenSource que más se ajuste a sus requerimientos.

2.3.1.3. Archivos de construcción

Tipo: Archivo

Propósito: Este archivo permite construir y automatizar ciertos procesos de un proyecto. A su vez, depende del lenguaje de programación e incluso de las preferencias del usuario. Para trabajar con C/C++, se utiliza comúnmente el archivo llamado Makefile, pero en Python se utiliza el archivo requeriments.txt, mientras que en Java se usa el archivo pom.xml.

2.3.2. Estructura de directorios

2.3.2.1. Python

Las aplicaciones Python tienden a usar una serie de librerías y módulos que no vienen incluidos en la librería estándar. Dicha restricción lleva a los desarrolladores a usar ambientes virtuales que permitan instalar versiones específicas de Python, así como de las librerías de interés para cada aplicación. Además del ambiente virtual, Python usa pip como gestor de paquetería para instalar de manera más eficiente las librerías y su versión específica. Dicha instalación queda identificada en el archivo *requriments.txt*.

En los ambientes locales es común usar plataformas de desarrollo como Anaconda, un sistema robusto para desarrollar código Python con todo tipo de ventajas. Sin embargo, cuando el código va a ser compartido y reposará sobre contenedores, es útil manejar directamente librerías de control en ambientes como pip, pyenv, virtualenv, entre otras.

La programación de Python viene gobernada por un conjunto de documentos llamado Python Enhancement Proposals, abreviado como PEP. A diferencia de lenguajes como Java que tienen un estilo muy estricto y definido principalmente por sus creadores (Oracle). Python ha venido evolucionando de la mano de sus desarrolladores y no existen reglas estrictas que indiquen unas

buenas prácticas estándar. Lo más cercano a este escenario es el documento PEP 8, donde su creador Guido van Rossum definió una serie de prácticas que todo desarrollador debería seguir. Es de resaltar que dicho documento se remonta al año 2001 y Python ha tenido muchos cambios desde aquella fecha.

Python a diferencia de otros lenguajes que siguen la filosofía propuesta por C/C++, tiene una estructura completamente diferente que agrupa su sistema de archivos en módulos y paquetes. Los módulos se entienden como archivos con extensión py que contienen alguna funcionalidad, mientras que los paquetes son directorios que agrupan múltiples módulos. Para que un paquete sea considerado por el intérprete simplemente es necesario que tenga dentro el archivo __init__.py.

Es preciso mencionar que Python también propone un estilo de nomenclatura para identificar sus ficheros. Por ejemplo, el nombre del fichero solo puede contener letras minúsculas y se debe utilizar el carácter guion bajo para separar palabras.

Así: app/data/my_settings.py

Así no: app/Data/mySetting.py

A continuación, la Figura 8 permite visualizar una estructura de la plantilla de un proyecto en Python:

Figura 8. Estructura de un proyecto en Python

Dado que los proyectos en Python vienen en estructuras, tamaños y objetivos muy diferentes, definir los entregables exactos para la documentación de un proyecto es bastante complejo. Dependiendo del tipo de proyecto la documentación tendrá unos u otros elementos.

Cada uno de estos archivos se usa según el tipo de proyecto: privado, compartido o libre (Open Source).

2.3.2.1.1. Proyecto privado

Son proyectos personales o de un par de personas. La mayor parte de estos proyectos recaen en el mismo desarrollador y su documentación es simple y concreta. Para estos proyectos se usa:

- README.
- examples.py

Estos proyectos no necesitan de un proceso de documentación riguroso, pero no debe desestimarse el uso de docstring o un buen archivo README para evitar que el proyecto sea incomprensible a futuro.

2.3.2.1.2. Proyecto compartido

Un proyecto compartido implica la participación de equipos pequeños donde existen metodologías de trabajo claras y conocimiento cercano entre los contribuyentes. Esta documentación es más rigurosa pues implica que el proyecto es de uso común. De igual manera, requiere tener una estructura bien definida para que nuevos contribuyentes se adapten al estilo de programación y puedan integrar sus aportes facilmente. Estos proyectos hacen uso de:

- README.
- examples.py
- HOW_TO_CONTRIBUTE o sección en README
- requeriments.txt

2.3.2.1.3. Proyecto de código libre

Son proyectos de alto impacto donde están inmersos múltiples equipos, desarrolladores e incluso compañías. Su prioridad máxima es la documentación y necesitan de un desarrollo claro, con metodologías sencillas capaces de adaptarse a múltiples flujos de trabajo. Los proyectos de código libre generalmente implican orden y una documentación de alta calidad con guías de uso. Además de facilidades para testear y proponer código. Este proyecto necesita de todos los archivos propuestos.

2.3.2.2. Java

Para proyectos que se realicen con el lenguaje de programación Java se propone usar el software para la gestión y comprensión de proyectos Apache MAVEN y mantener su estructura inicial como lo muestra la Figura 9.

Los desarrollos Java tienen la peculiaridad de estar soportados por un ambiente empresarial robusto que lleva años cuidando los detalles de todo tipo de proyectos. La estructura MAVEN garantiza aislamiento entre capas y separación funcional para mantener buenas prácticas de

desarrollo. El control por paquetes permite importar fácilmente otras clases y garantizar abstracción en la inyección de dependencias.

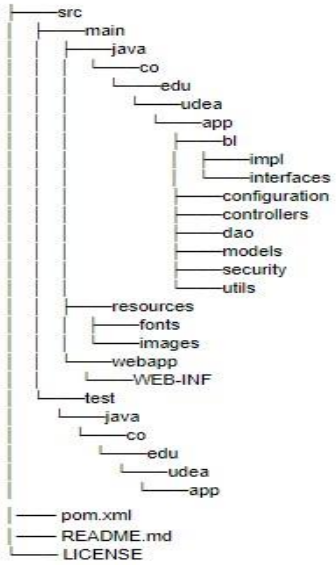

Figura 9. Estructura de un proyecto en Java con Maven

2.4. Documentación

Esta sección busca plasmar algunas recomendaciones para documentar código en proyectos de software al interior del grupo de investigación GITA. Sus principios están diseñados para evitar reprocesos y mejorar la mantenibilidad del código. En resumen, esta sección busca que el desarrollo al interior del grupo sea transversal a la documentación y exista un balance entre los tiempos de entrega y el apoyo a la mantenibilidad del proyecto.

2.4.1. Documentación en Python

Python desde su versión 3.5 nos brinda la posibilidad de detallar nuestra documentación mediante *Type Hinting*, una solución estática para indicar el tipo de un valor en Python. *Type Hinting* fue especificado en el PEP 484 e introducido en Python 3.5.

Es de resaltar que para el desarrollador diseñar y explicar partes del código sin hacer comentarios es esencial.

Mediante el Código 1 podemos identificar que la función recibe un parámetro de entrada que es de tipo string y otro de salida que igualmente es de tipo string. El nombre de la propia función nos contextualiza para identificar que su propósito es retornar/presentar un mensaje de saludo.

```
1. def hello_world(name: str) -> str:
```

```
2.          return(f"Hello {name}")

3.
```

Código 1. Hola mundo en Python

En Python todos los objetos cuentan con una variable especial llamada doc gracias a la que podemos describir para qué sirven los y cómo se usan los objetos. Estas variables reciben el nombre de docstrings, cadenas de documentación.

En Python todo es un objeto, por ende, cada clase que usamos hereda de *object*. Si tratamos de acceder al atributo __doc__ de un objeto en Python podemos visualizar que allí se encuentra la documentación definida para dicho objeto. Mediante Código 2 es posible visualizar en Python la documentación de un objeto.

```
1. >>> help(int)

2. >>> print(int.__doc__)
```

Código 2. Acceso a la documentación del objeto int mediante la función help

Si queremos documentar nuestro código podemos recurrir al Código 3.

```
1. def hello_world(name: str) -> str:

2.         return (f"Hello {name}")

3. hello_world.__doc__ = 'Docs'
```

Código 3. Ejemplo de documentación de una función

Como puede identificarse al seguir el anterior ejemplo, esta metodología de documentación es bastante tediosa y lenta. Sin embargo, Python mediante su compilador nos brinda opciones más limpias y eficientes con las que podemos documentar el código. Simplemente con ubicar una cadena de caracteres en cierta posición de un objeto, el compilador las identifica y las asocia a la variable __doc__ automáticamente (ver Código 4).

```
1. def say_hello(name):

2.        """Una funcion simple que permite saludar."""

3.        print(f"Hello {name}, is it me you're looking for?")
```

Código 4. Documentación de una función mediante docstring

2.4.1.1. Tipos de DocString

Las convenciones de doctring se describen en el PEP 257 y su propósito es proporcionar a los usuarios una breve descripción general del objeto. Esta descripción debe ser lo suficientemente concisa para que sea fácil de mantener, pero aun así ser lo suficientemente clara para que los nuevos usuarios entiendan su propósito. En todos los casos, los docstring deben usar el formato de cadena de comillas doble con triple repetición. Esto se debe hacer tanto si la documentación es de varias líneas o no, aunque como mínimo un *docstring* debe tener un resumen rápido de una línea.

Las cadenas de documentos con múltiples líneas se utilizan para hablar más sobre el objeto, es decir, dan a conocer un resumen de este. Todas las cadenas de documentos con múltiples líneas tienen las siguientes partes:

- Una línea de resumen de una línea.
- Una línea en blanco que procede del resumen.
- Información del objeto.
- Otra línea en blanco

Según el tipo de objeto en Python el proceso de documentación es diferente. Los principales objetos que se documentan en Python son: scripts, paquetes, clases, funciones y variables.

2.4.1.2. Documentación para clases

La documentación de una clase debe hacerse tanto para la estructura general de la clase como para los elementos que ésta incorpora, si éstos son públicos. Para documentar una clase se debe tener en cuenta:

- Un breve resumen del propósito de la clase.
- Los datos de la persona que la crearon e información de contacto.
- Una breve descripción en cada uno de los métodos públicos.
- Información sobre las propiedades (atributos) públicos en la clase.

Los métodos de una clase deben documentarse de manera individual, tratando de ser lo más claros posibles. El único método con documentación especial es __init__ en este solo se deben documentar los parámetros que recibe, pues el proposito de dicho método es siempre el mismo. Al documentar los métodos se debe tener en cuenta:

- Una breve descripción de qué es el método y para qué se utiliza.
- Documentar cualquier argumento (tanto obligatorio como opcional).
- Etiquetar cualquier argumento que se considere opcional o que tenga un valor predeterminado.
- Información sobre las propiedades (atributos) públicos en la clase.

- Cualquier efecto secundario que ocurra al ejecutar el método (Excepciones).
- Cualquier restricción sobre el método

El Código 5 permite identificar la manera correcta de realizar comentarios en el código Python para una clase particular.

```
1.  class Router:
2.      """
3.          Clase usada para describir un router
4.
5.          ...
6.
7.          Attributes
8.          ----------
9.          name : str
10.             Nombre que para reconocer el Router
11.         brand : str
12.             Marca a la cual pertenece el router
13.         hosts : list
14.             Lista de hosts conectados al router
15.
16.         Methods
17.         -------
18.         def_host(host: Host)
19.             Asigna una posicion de la tabla de enrutamiento del router al host - IP
20.         received_packet(packet: Packet)
21.             Envia el paquete al host
22.         """
23.     table = {}
24.
25.     def __init__(self, name, brand, hosts):
26.         self.name = name
27.         self.__brand = brand
28.         self.hosts = [self.def_host(host) for host in hosts]
29.
30.     def def_host(self, host):
31.         """ Asigna una posicion de la tabla de enrutamiento del router al host - IP
32.
33.         Parameters
34.         ----------
35.         host : Host
36.             Indica el host que se desea agregar a la tabla de enrutamiento.
37.
38.         Raises
39.         ------
40.         NotImplementedError
41.             Si el paquete enviado no es del tipo BasePacket, el sistema retorna error
42.
43.         """
44.         self.table[host.ip] = host
45.
46.     def received_packet(self, packet):
47.         """ Captura el paquete enviado por el host y lo procesa
48.
49.         Parameters
50.         ----------
51.         packet : BasePacket
52.             Paquete que contiene la informacion
53.
54.         Raises
55.         ------
56.         NotImplementedError
57.             Si el paquete enviado no es del tipo BasePacket, el sistema retorna error
58.
59.         """
60.         if isinstance(packet, BasePacket):
61.             raise NotImplementedError("Error en el paquete")
62.         self.table[packet.dst].received(packet)
```

```
63.
64.
```

Código 5. Clase documentada

2.4.1.3. Documentación para paquetes

La documentación de un paquete debe colocarse en la parte superior del archivo __init__.py del paquete. El doctring debe enumerar los módulos y subpaquetes que exporta el paquete. Los docstring de un paquete son similares a los docstring de una clase. Mientras que en la clase se documentan los atributos y los métodos, en el paquete se documenta los módulos y las funciones que se encuentran dentro.

2.4.1.4. Documentación para scripts

Al ser Python un lenguaje basado en scripts, estos son usados ampliamente en la comunidad para un sinfín de tareas. En caso de tener que documentar un script es necesario ser muy precisos con el comportamiento en tiempo de ejecución. También es necesario documentar cada uno de los parámetros que se requieren para funcionar y, sobre todo, los posibles errores que pueden resultar en una ejecución indebida. Finalmente, cualquier importación personalizada o de terceros debe ser enumerada dentro del docstring.

2.4.1.5. Documentación para funciones y variables.

Las funciones y variables se documentan igual que los métodos y atributos de una clase. Es de resaltar que los métodos son funciones dentro de una clase y los atributos son variables dentro de una clase.

2.4.1.6. Formatos para documentar en Python

A diferencia de otros lenguajes de programación, la comunidad de Python no tiene definida una estructura única para implementar el texto dentro de los docstring. Sin embargo, existen algunas implementaciones famosas que están ampliamente extendidas entre los desarrolladores de Python. El Cuadro 2 resume los formatos de documentación más conocidos [25].

Formatos para documentación en Python		
Formato	**Descripción**	**Especificación formal**
Google docstrings	Recomendación de Google para	No

	documentar código en Python	
reStructured Tex	Documentación propuesta por Python “Oficial”	Si
NumPy/SciPy docstrings	Documentación propuesta por NumPy que une la documentación de Google con la documentación oficial.	Si
Epytext	Adaptación de la librería EpicDoc para Python	Si

Cuadro 2. Cuadro con los formatos de documentación en Python

2.4.2. Documentación en Java

La documentación en Java está enfocada en la creación de documentos altamente estructurados y de fácil creación. Debido a que Java se encuentra enfocado al ambiente empresarial, los comentarios adquieren mayor importancia que en otros lenguajes de programación.

Hay dos formas diferentes de escribir comentarios en Java: como especificaciones de la API y como documentación de la guía de programación. Una empresa con mayor capacidad económica puede permitirse combinar ambos en la misma documentación (debidamente fragmentada).

En Java la herramienta para documentar el código se denomina JavaDocs, esta herramienta viene implementada en el core del lenguaje de programación como una de sus APIs.

2.4.2.1. JavaDocs

Es una API para documentar el código de Java que hace parte del paquete de código genérico (JDK, Java Development Kit). Esta API convierte los comentarios en archivos asociados a la documentación de Java, los cuales son usados para promover las buenas prácticas de desarrollo. JavaDocs permite generar 4 tipos de archivos diferentes:

- **Archivos de código fuente para clases Java (.java**): Contienen comentarios de clase, interfaz, campo, constructor y método.
- **Archivos de comentarios de paquetes:** Estos contienen comentarios de paquetes
- **Resumen de archivos de comentarios:** Estos contienen comentarios sobre el conjunto de paquetes
- **Varios archivos sin procesar:** Estos incluyen imágenes, código fuente de muestra, archivos de clase, applets, archivos HTML.

2.4.2.2. ¿Cómo escribir comentarios en Java?

Un comentario en Java está escrito en HTML (Hypertext Markup Language) y debe preceder a una declaración de clase, campo, constructor o método. Los comentarios deben comenzar con una barra inclinada y dos asteriscos (/ ** ... * /), y pueden incluir etiquetas especiales para describir características como parámetros de métodos o valores de retorno. Ver Código 6, [26].

2.4.2.3. Etiquetas de bloque JavaDocs

Las etiquetas más usadas para los JavaDocs son:

- **@author:** Describe quien es el autor del código del software desarrollado.
- **@version:** Agrega el subtítulo versión al JavaDocs con la versión del código usado
- **@param**: Da a conocer la información de un parámetro definido en un método.
- **@return:** Indica que la función retorna un dato especifico.
- **@exception/@throws:** Indica que la clase retorna algún tipo de excepción.
- **@see:** Nos permite generar un enlace a una clase usada dentro del elemento a documentar.
- **@since** Especifica la fecha o versión desde que la funcionalidad se encuentra disponible.
- **@deprecated:** Indica que un elemento ya no hace parte de la funcionalidad actualizada de la aplicación. Por lo general los elementos en desuso tiene un reemplazo.

```
/**
 * Devuelve un archivo ubicado en una base de datos
 * El argumento url debe especificar una {@link URL} absoluta. El nombre
 * argumento es un especificador que es relativo al argumento url.
 *
 * @author JavaSoftware
 * @param URL absoluto donde se encuentra el archivo
```

```
* @param  name Ubicacion del archivo respecto a la URL
* @return      Archivo de la URL
* @see         byte
*/
public byte[] getFile(URL url, String name) {
        try {
            return searchData(new URL(url, name));
        } catch (MalformedURLException e) {
            return null;
        }
}
```

Código 6. Código de ejemplo para documentación en Java

2.4.2.4. Consejos para documentar en Java

- **Use @link y @linkplain para señalar algún código.** Cuando existen dependencias entre clases, la manera más efectiva de mejorar la documentación es usando @Link. Esta etiqueta crea una referencia a la clase dependiente y facilita el acceso entre clases en la documentación final.
- **Usar @code para ejemplos dentro del código.** A menudo se puede encontrar algún código dentro del Javadoc para ilustrar cómo usar un método, clase o para proporcionar algún ejemplo. Para mostrar el código correctamente y evitar que se interpreten algunas marcas se puede usar @code.
- **Usar @value para insertar el valor de un campo en la documentación**: Cuando se necesita mostrar una constante del código en la documentación, ésta puede adicionarse mediante la etiqueta @value apuntando al nombre de la variable.
- **Usar la etiqueta @author para evitar el anonimato**: La etiqueta de autor es bastante útil para reconocer quienes son los responsables de un código especifico. Esta práctica ayuda a brindar crédito a los desarrolladores encargados de un proyecto específico.
- **Siempre usar la etiqueta @return y @param**: En Java gran parte del desarrollo se realiza mediante el IDE de preferencia personal. Sin embargo, cuando se importa una nueva librería es necesario conocer de manera rápida como implementar los métodos. Para detalles más específicos es necesario recurrir a la documentación completa. Sin embargo, la vista previa mostrada por los IDE es de bastante utilidad y acelera el desarrollo del código.

2.5. Sistema de control de versiones

Con los archivos ordenados es necesario tener un sistema que nos permita controlar el cambio de estos. En el proyecto se necesita mantener orden y trazabilidad para identificar errores y tener código reusable y mantenible. Para este fin se utilizan los sistemas de control de versiones como Git, su exponente más reconocido.

2.5.1. ¿Por qué Git?

Git es un Software de control de versiones no centralizado, orientado al mantenimiento de versiones de aplicaciones con ficheros de código. En la actualidad es, probablemente, el sistema de control de versiones más utilizado. En el pasado, Subversión adquirió cierta importancia (igualmente Mercurial o Bazaar). Sin embargo, Git se ha impuesto con fuerza por los siguientes motivos:

- Su eficiencia, eficacia y sencillez.
- Su capacidad de ser usado de forma básica por los nuevos usuarios.
- Su capacidad de ser usado de formas muy complejas y elaboradas por expertos.
- Su potencial, cantidad de prestaciones y control sobre el proyecto.

2.5.2. Sistema de alojamientos de repositorios Git

Son aplicaciones web de terceros que encapsulan y mejoran un sistema de control de versiones. No se puede utilizar por completo un servicio de alojamiento de repositorios sin tener que usar un sistema de control de versiones subyacente [27].

Los servicios de alojamiento de repositorios de códigos son todos parecidos en sus ofertas de nivel de superficie. Sin embargo, el único sistema OpenSource para alojamiento de repositorios es GitLab. Por esta razón, en la siguiente sección solamente se hablará sobre este.

2.5.2.1. ¿Por qué GitLab?

GitLab, a su vez, es un repositorio de gestión de proyectos dotado de interfaz web. Como podemos deducir del nombre, está construido sobre Git y nos proporciona el código para generar un servidor, gestionar los clientes, sus opciones y los servicios ofrecidos. A través de GitLab, podemos gestionar grupos, personas y los permisos que requieran los usuarios dentro de los grupos o proyectos a los que pertenezcan. También nos permite llevar a cabo un seguimiento del estado actual e histórico de los proyectos. Como software libre y gratuito, cuenta con una buena comunidad que lo mejora y actualiza constantemente. Las funcionalidades que este software ofrece son:

- **Opción de autentificar contra servicios como LDAP:** Un punto interesante, ya que otros servicios similares a GitLab no ofrecen esta opción de autentificación.
- **Distintos tipos de acceso y permisos (uso de roles y grupos):** Restringe proyectos a ciertos usuarios. También puede limitar el acceso al contenido, además de ciertas acciones concretas. Es de resaltar que, entre otros medios de acceso, los usuarios también pueden acceder al proyecto a través de la web o vía SSH (Secure Shell).
- **Seguimiento de incidencias y comentarios de un proyecto**: A partir de la interfaz web, los usuarios podrán comentar aspectos del proyecto que vean conveniente discutir. Se ofrece un servicio de tiqueting para hacer el seguimiento de incidencias u objetivos del proyecto y permite habilitar un wiki para la documentación que se requiera.
- **Código del servidor fácilmente accesible remotamente:** Al trabajar con un servidor propio, se puede asegurar la conexión desde el exterior, aislando así un punto de dependencia respecto a un servicio externo. Al tratarse de un servicio propio, se pueden activar filtros para limitar el acceso a un rango de redes particular.
- **Gestión de grupos y proyectos:** Permite gestionar proyectos y grupos, controlando los permisos y la libertad de los usuarios dentro del sistema. Estas funciones son prácticas para poder trabajar en equipo con conjuntos de usuarios sin tener que definir restricciones individuales. Esto permite una mayor facilidad, flexibilidad y rapidez ante cambios de la administración del proyecto.
- **Capacidad para importar repositorios ya existentes**: Puesto que GitLab es un sistema relativamente nuevo y mucha gente ha trabajado ya en otros sistemas como GitHub o BitBucket, nos permite crear repositorios a partir de otros ya creados con anterioridad.
- **Copias de seguridad**: Al estar ubicado en el servidor, existen copias de seguridad locales. Si se perdiera alguna parte del proyecto, se dispone de copias del sistema que no hayan sido comprometidas.
- **Historial de modificaciones del proyecto**: Es especialmente práctico en trabajos en grupo debido a que, cuando se hace alguna modificación, se puede identificar clara e intuitivamente dichos cambios.

2.5.3. Flujos de trabajo en Git

Para comenzar a desarrollar sobre los sistemas de control de versiones es necesario definir un flujo de trabajo idóneo que nos permita mantener trazabilidad sobre un proyecto con el paso del tiempo. El flujo de trabajo define un modelo de ramificación diseñado en torno al lanzamiento del proyecto. A su vez, proporciona un marco robusto para gestionar proyectos más grandes y complejos. Uno de los flujos de trabajo en Git más conocidos es GitFlow, en éste se concentrará el trabajo de grado.

2.5.3.1. GitFlow

Gitflow es un flujo de trabajo utilizado en el sistema de control de versiones Git. En éste, se le da especial importancia a la interacción entre apuntadores de capturas instantáneas del estado del código que permiten mantener la trazabilidad de un proyecto. Es decir, la interacción entre ramas.

Este flujo de trabajo no agrega nuevos conceptos o comandos más allá de lo que se requiere para el Flujo de trabajo de la rama de funciones. En cambio, asigna roles muy específicos a diferentes ramas y define cómo y cuándo deben interactuar. Además de las ramas de características, utiliza ramas individuales para preparar, mantener y grabar lanzamientos [28].

2.5.3.1.1. Rama desarrollo (develop) y maestra (master)

Como se puede visualizar en la Figura 10 en vez de usar una sola rama *master*, este flujo de trabajo usa dos ramas para guardar el historial de estados del proyecto. La rama *master* contiene todo el código oficial que se encuentra verificado y la rama de desarrollo contiene todo el código que se encuentra en desarrollo o está siendo probado. A medida que más funcionalidades se encuentran listas en la rama de desarrollo se debe integrar la información de la rama de desarrollo a la rama *master*. Sin embargo, esta integración no se debe realizar directamente, sino se debe realizar mediante la rama Release, la cual será explicada más adelante.

La integración debe ser identificada mediante una versión para poder llevar un mejor registro de los cambios realizados. La versión se identifica mediante un *TAG* que lleva el *commit* de *merge* (integración) [28].

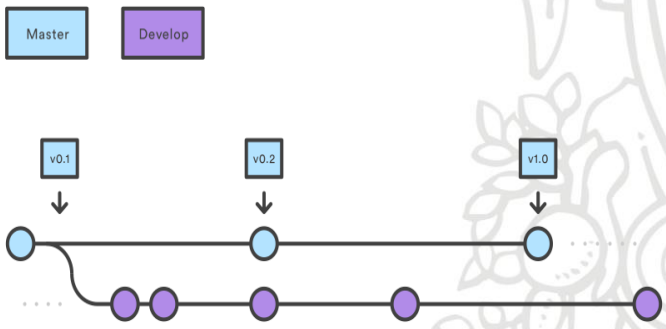

Figura 10. Rama maestra y rama de desarrollo, metodología GitFlow

2.5.3.1.2. Ramas de funcionalidad

En la Figura 11 se puede apreciar que cada nueva característica debe residir en su propia rama, la cual, se puede enviar al repositorio central para respaldo/colaboración. Estas ramas en lugar de bifurcarse del maestro usan la rama de desarrollo como su rama principal. Cuando se completa una característica, se fusiona nuevamente con la rama de desarrollo. Las ramas de funcionalidad nunca deberían interactuar directamente con rama *master* [28].

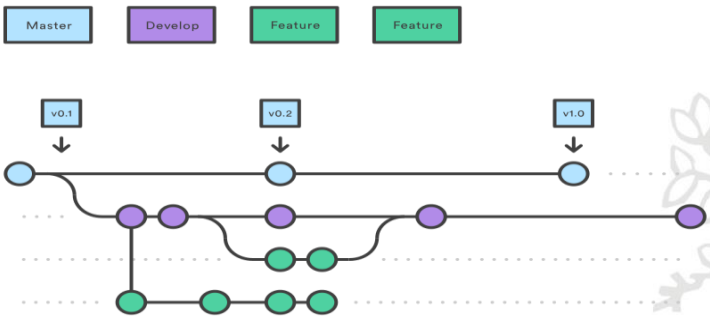

Figura 11. Rama de funcionalidad usada en GitFlow

2.5.3.1.3. Ramas de lanzamiento (*release*)

Una vez que el desarrollo ha adquirido suficientes características para un lanzamiento (o se acerca una fecha de lanzamiento predeterminada), se debe bifurcar una rama de lanzamiento fuera de la rama de desarrollo. La creación de esta rama inicia el siguiente ciclo de lanzamiento, por lo que no se pueden agregar nuevas características después de este punto. Solo las correcciones de errores, la generación de documentación y otras tareas orientadas a la versión deben ir en esta rama.

Una vez que la rama esta lista para enviarse, como muestra la Figura 12, la rama de lanzamiento se fusiona en la rama *master* y se etiqueta con un número de versión. Además de esto, la rama debe fusionarse nuevamente en la rama de desarrollo, que puede haber progresado desde que se inició el lanzamiento. El uso de una rama dedicada para preparar versiones hace posible que un equipo pueda pulir la versión actual. Al igual que las ramas de características, las ramas de lanzamiento se basan en la rama de desarrollo [28].

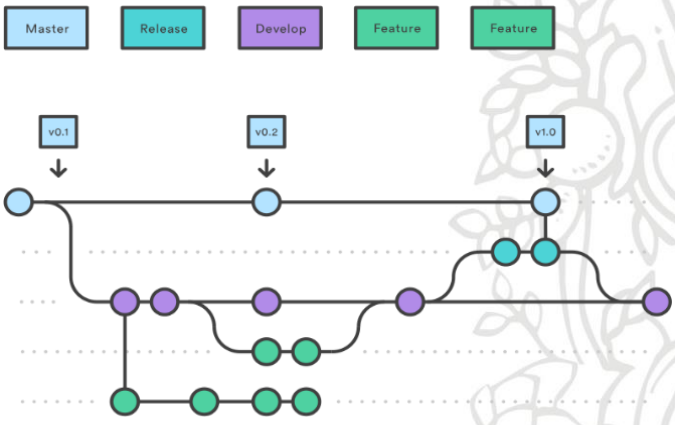

Figura 12. Rama de lanzamiento usada en GitFlow

2.5.3.1.4. Ramas de mantenimiento (HOTFIX)

Se utilizan para parchear rápidamente lanzamientos de producción. Las ramificaciones de revisión son muy parecidas a las ramificaciones de lanzamiento y ramificaciones de características, excepto que se basan en la rama *master* en lugar de la rama de desarrollo. Como se puede visualizar en la Figura 13 ésta es la única rama que debe bifurcarse directamente de *master*. Tan pronto como se complete la corrección, debe fusionarse tanto en la rama *master* como en la rama de desarrollo (o en la rama de la versión actual), y el *commit* en la rama *master* debe etiquetarse con un número de versión actualizado [28].

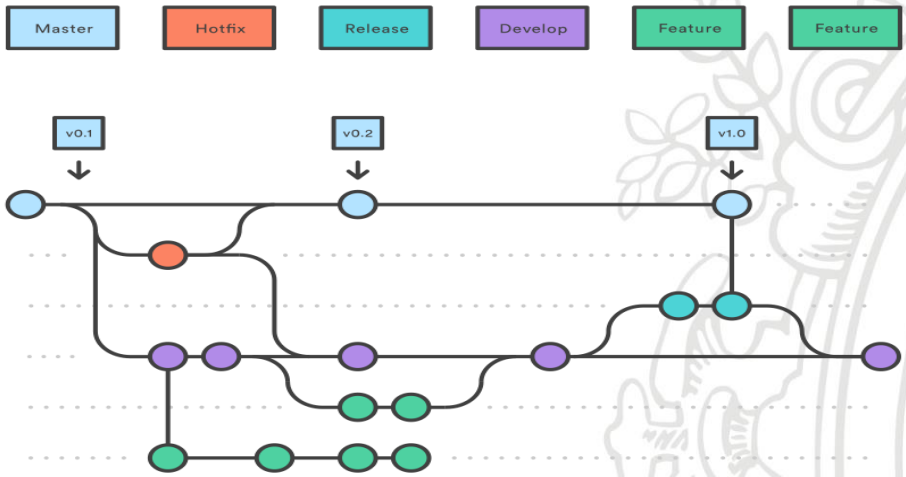

Figura 13. Rama de mantenimiento en GitFlow

2.6. Pruebas de software

Hay muchas maneras de probar código, conforme pasa el tiempo se han desarrollado diferentes metodologías y herramientas que facilitan dicho proceso. Es probable que todo desarrollador de código haya realizado una prueba en algún momento. Dado que el simple hecho de ejecutar una aplicación y verificar su funcionamiento se considera como una. Este tipo de pruebas se llaman pruebas exploratorias y son el concepto de donde nace todo el mundo de las pruebas.

2.6.1. Pruebas de software en Python

Una visualización general de las pruebas en Python nos permite identificar, a rasgos generales, que éstas se resumen en dos preguntas ¿Cómo probar código en Python? Y ¿Cómo implementar código en Python? La primera pregunta condensa las herramientas para poder realizar las pruebas y todos los temas que se desprenden de estas. La segunda resuelve el tema de cómo pensar para llevar a cabo las pruebas y todo lo que se requiere para ejecutar una prueba especifica. En la Figura 14 se puede visualizar el mapa general de pruebas en Python y las correspondientes secciones.

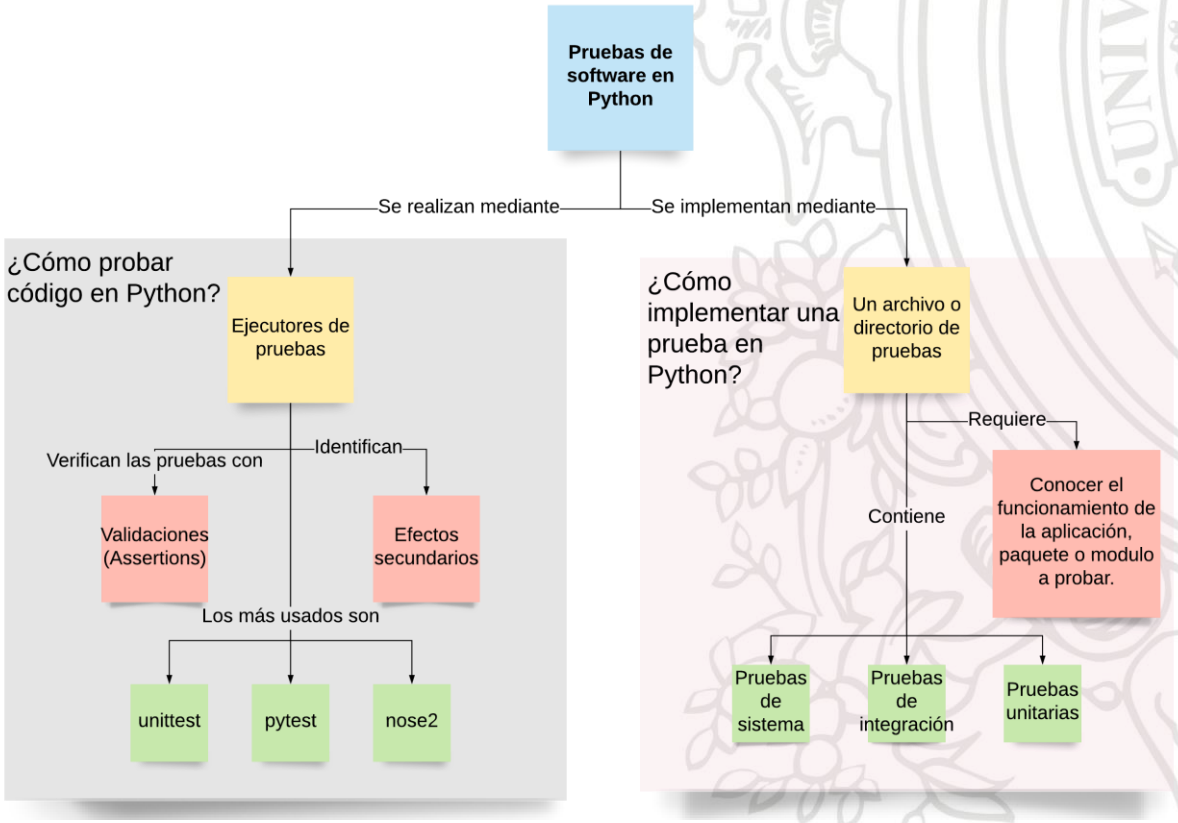

Figura 14. Mapa conceptual que resumen pruebas en Python

2.6.1.1. ¿Cómo probar código en Python?

En Python, para realizar pruebas, se utilizan librerías que se denominan ejecutores de pruebas. Python tiene una gran cantidad de ejecutores, pero el oficial es unittest. A pesar de esto, existe la posibilidad de escoger otros ejecutores dependiendo de las necesidades y gustos del desarrollador. Algunos de los factores que pueden influir para elegir un ejecutor sobre otro son:

- El tamaño del proyecto.
- El paradigma de programación.
- La clase de pruebas a realizar.
- La metodología de desarrollo

Adicional a esto se tiene que todo ejecutor usa funciones llamadas asserts para verificar que la respuesta obtenida en la prueba sea la esperada por el desarrollador. Todas las herramientas del ejecutor relacionadas con dichas funciones se denominan validaciones (Assertions) y cada ejecutor tiene una manera diferente de implementarlas.

Los tres ejecutores de pruebas más usados en Python son:

- unittest
- nose2
- pytest

En el Cuadro 3, se hará una comparación entre los ejecutores más relevantes en Python.

Ejecutor de pruebas	**unittest**	**pytest**	**nose2**
Ventajas	- Es fácil de usar - Tiene soporte por parte del equipo principal de Python. - Es muy estable.	- Permite el uso de funciones como pruebas. - Soporta el comando assert nativo de Python. - Permite controlar casos de error en las pruebas. - Tiene un amplio ecosistema de plugins.	- Es fácil de usar. - Excelente sistema para organizar archivos de prueba. - Sintaxis clara. - Soporta el comando assert nativo de Python
Desventajas	- Usa métodos espaciales para validar las pruebas. - Es poco parametrizable.	- La curva de aprendizaje es alta. - Es poco usada en el ámbito OpenSource.	- Es complicado migrar entre nose y nose2. - La documentación no es apta para principiantes. -

	- No permite inyectar dependencias.	- No cuenta con soporte oficial de Python.	
Casos de uso	- Proyectos OpenSource donde colaboran muchos usuarios. - Proyectos que se implementen en la biblioteca nativa de Python. - Proyectos para desarrolladores con poca experiencia. - Proyectos que usan programación imperativa.	- Proyectos grandes y medianos. - Proyectos con lógica de negocio complicada. - Proyectos que necesiten parametrizar las pruebas por ambientes. - Proyectos que usan programación funcional.	- Proyectos grades, medianos y pequeños. - Proyectos con múltiples tipos de pruebas. - Proyectos que le dan mucha importancia a las pruebas.
Calidad de documentación	Excelente	Regular	Regular
Soporta combinación con unittest	No aplica	Si	Si
Instalación	No necesita, viene en la biblioteca estándar de Python.	Mediante PyPI	Mediante PyPI
Popularidad	Alta	Media	Alta

Cuadro 3. Comparativa entre los ejecutores de pruebas en Python

2.6.1.1.1. Validaciones

Las validaciones en Python son un conjunto de herramientas (entiéndase para este caso como herramientas: a funciones, variables o instrucciones) que sirven probar condiciones en las pruebas. Algunas buenas prácticas respecto a las validaciones de resultados son:

- Se debe asegurar que las pruebas sean repetibles y que al ejecutarse varias veces se obtengan los mismos resultados.
- Se debe garantizar que los resultados se relacionen con los datos de entrada.
- Las validaciones no deben contener excepciones no contempladas.
- Las validaciones no deben usarse en tiempo de ejecución, es decir, solo se deben utilizar en las pruebas.
- Los datos de entrada en la función de validación deben ser íntegros, es decir, no deben ingresarse otro tipo de datos que no sean booleanos.

En el siguiente cuadro se compara algunos de los métodos de validación usados en los ejecutores de pruebas.

unittest	**pytest y nose2**	**Equivalente**
assertEqual(a, b)	assert a == b	a == b
assertTrue(x)	assert bool(c)	bool(x) is True
assertFalse(x)	assert not bool(c)	bool(x) is False
assertIs(a, b)	assert a is b	a is b
assertIsNone(x)	assert x is None	x is None
assertIn(x)	assert a == b	a in b
assertIsInstance(x)	assert a == b	isinstance(a, b)
assertRaises(Ex)	assert a == b	raise Ex

Cuadro 4. Comparativa de los métodos para validación en pytest, unittest y nose2

2.6.1.1.2. Efectos colaterales o secundarios

La verificación de un código no siempre es tan simple como mirar el valor de retorno de una función. A menudo, el cambio de un fragmento de código puede alterar otras funciones en el sistema, como el atributo de una clase, un archivo o un valor en una base de datos. Estos se conocen como efectos secundarios y son una parte importante de las pruebas.

Si la unidad de código que desea probar tiene muchos efectos secundarios, es posible que se esté rompiendo el **principio de responsabilidad única**. Romper este principio significa que el código está haciendo demasiadas cosas y sería mejor ser refactorizado. Seguir este principio es una excelente manera de diseñar código que permita escribir pruebas unitarias repetibles. Lo cual se traduce en aplicaciones confiables.

Para la siguiente sección se utilizará como librería ejecutora de pruebas unittest, dado que es una de las librerías más populares y viene instalada por defecto en la biblioteca estándar de Python.

2.6.1.2. ¿Cómo implementar una prueba en Python?

Para tener una idea de cómo podría implementarse una prueba pensemos en el siguiente escenario ¿Qué pasa si el computador no inicia? ¿Dónde está el error? En casos como estos es donde las pruebas toman importancia y cobra sentido entender el enfoque de cada una.

Inicialmente el toque al botón de encendido es una prueba de integración, pues nos permite reconocer si existe un problema, más no nos permite determinar el punto exacto de la falla. En palabras simples una prueba de integración verifica el componente y su operabilidad con el resto de los elementos. Sin embargo, si se desea reconocer el punto exacto de falla, es necesario probar individualmente cada componente, es decir, realizar pruebas unitarias.

Para contextualizar el uso de pruebas usaremos un ejemplo basado en escenario anterior.

En la Figura 15 podemos identificar la organización un paquete llamado *pc*. La primera pregunta que podría surgir al ver una estructura de este tipo es ¿Dónde y cómo debo localizar las pruebas?

Para las pruebas es una convención asegurar que cada archivo comience con el prefijo "test". De tal manera que todos los corredores de pruebas en Python asuman que dicho archivo contiene pruebas para ser ejecutadas. Además, los archivos de prueba siempre deben ubicarse al mismo nivel del directorio que contiene el paquete de Python que será evaluado. También es importante aclarar que, para algunos proyectos muy grandes, es más útil dividir las pruebas en subdirectorios según su propósito o uso.

Figura 15. Estructura de directorios para el ejemplo Calculator

En la Figura 16 se puede visualizar un ejemplo de cómo debería ubicarse y nombrar un archivo de pruebas en Python.

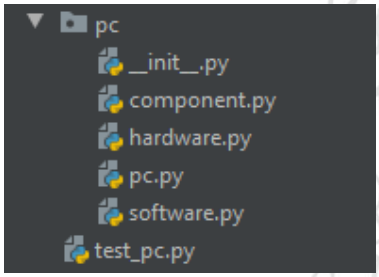

Figura 16. Estructura de directorios con archivo de pruebas

La Figura 17 resume la información de la estructura de paquete pc. En ésta, las listas, las variables estáticas, las variables de instancia y los métodos de clase se encuentran codificados por color, mientras las clases y módulos no. Esto se debe, a que el diagrama busca mostrar

claramente la diferencia entre los módulos y la jerarquía de cada uno. Mientras con los demás elementos el diagrama busca que el lector identifique las similitudes entre los elementos internos de las clases.

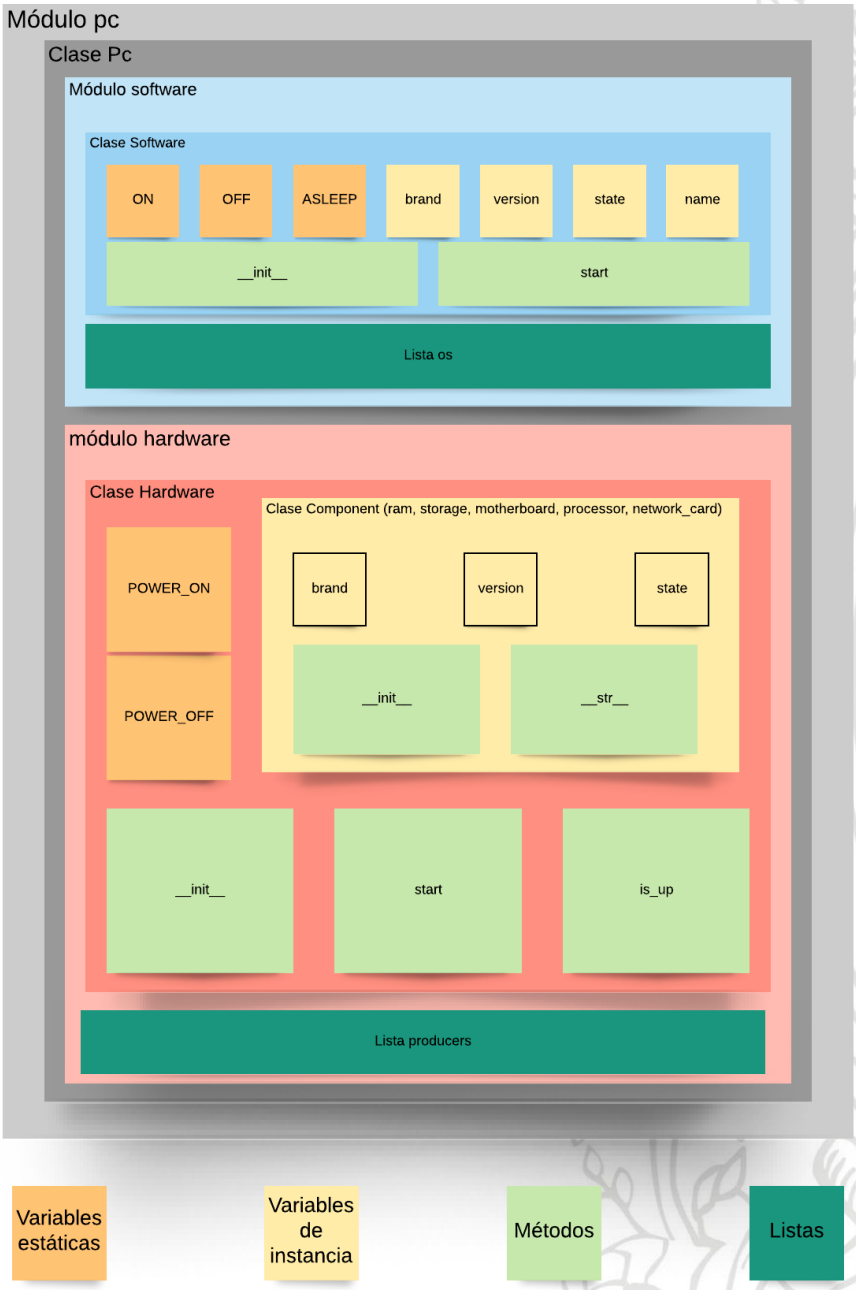

Figura 17. Diagrama de estructura del paquete pc

Los siguientes códigos nos permiten visualizar los módulos que conforman el paquete pc. Cabe destacar que se debe reconocer la funcionalidad de cada módulo antes de plantear las pruebas.

Debido a que esto nos permite tener claridad sobre que se desea probar y si funciona o no de manera correcta.

El módulo más pequeño y usado es *component*. Dicho modulo se encuentra en el Código 7 y contiene una clase llamada Component la cual posee los siguientes atributos: state, brand y name. Además, cuenta con un constructor que inicializa sus atributos y una función para imprimir un mensaje personalizado al usar la función print de Python con un objeto de la clase Component.

```
1.  class Component:
2.
3.      def __init__(self, brand, name):
4.          self.state = None
5.          self.brand = brand
6.          self.name = name
7.
8.      def __str__(self):
9.          return f'Yo soy {self.name} diseñado por {self.brand} '
10.
```

Código 7. Clase Component del módulo component

En el Código 8 se puede apreciar el módulo hardware. Este cuenta con dos elementos, una clase llamada Hardware y una lista de Strings llamada producers. La lista tiene nombres de fabricantes de hardware. La clase contiene dos variables estáticas llamadas POWER_OFF y POWER_ON que sirven como constantes para simbolizar los estados del hardware. También contiene 5 variables de instancia de tipo Component que sirven para representar los componentes que tiene el hardware y un constructor que las inicializa. Para finalizar, es posible visualizar un último método llamado *start*, el cual sirve para encender cada uno de los componentes del hardware y ponerlo en modo POWER_ON.

```
1.  from pc.component import Component
2.
3.
4.  class Hardware:
5.      POWER_OFF = 'POWER OFF'
6.      POWER_ON = 'POWER ON'
7.
8.      def __init__(self, ram: Component = None, storage: Component = None,
9.                   network_card: Component = None, processor: Component = None,
10.                  motherboard: Component = None):
11.         self.ram = ram
12.         self.storage = storage
13.         self.network_card = network_card
14.         self.processor = processor
```

```
15.         self.motherboard = motherboard
16.         self.state = self.POWER_OFF
17.
18.     def start(self):
19.         try:
20.             self.ram.state = self.POWER_ON
21.             self.storage.state = self.POWER_ON
22.             self.network_card.state = self.POWER_ON
23.             self.processor.state = self.POWER_ON
24.             self.motherboard.state = self.POWER_ON
25.             self.state = self.POWER_ON
26.         except AttributeError:
27.             self.state = self.POWER_OFF
28.             raise ValueError('Hardware doesnt start')
29.
30.     def is_up(self):
31.         return self.state == self.POWER_ON
32.
33.
34. producers = ['APPLE', 'ASUS', 'SAMSUNG', 'MICROSOFT', 'IBM', 'INTEL']
35.
```

Código 8. Módulo hardware

En el Código 9 podemos identificar el módulo software. Cuenta con dos elementos, una clase llamada Software y una lista llamada os. La lista contiene el nombre de tres sistemas operativos. La clase contiene tres variables estáticas llamadas ON, OFF y ASLEEP que sirven como constantes que simbolizan un estado. También contiene cinco variables de instancia que son inicialadas en el constructor y son: brand, name, versión, state y hardware. Para finalizar contiene un método llamado start, el cual sirve para encender el sistema operativo, y a su vez depende de que el hardware este encendido.

```
1.  from pc.hardware import Hardware, producers
2.
3.
4.  class Software:
5.
6.      ON = 'STARTED'
7.      OFF = 'SHUTDOWN'
8.      ASLEEP = 'ASLEEP'
9.
10.     def __init__(self, brand, name, version, hardware: Hardware):
11.         self.brand = brand
12.         self.name = name
13.         self.version = version
14.         self.state = self.OFF
```

```
        if not isinstance(hardware, Hardware):
            raise TypeError("Software need a hardware")
        else:
            self.hardware = hardware
        if self.name == os[1] and (hardware.processor != producers[0]):
            raise TypeError("Apple only use Apple processors")

    def start(self):
        if self.hardware.is_up():
            self.state = self.ON
        else:
            raise ValueError("Software cant start without hardware")


os = ['WINDOWS', 'MAC', 'LINUX']
```

Código 9. Módulo software

Cómo último modulo tenemos a pc, el cual se encuentra en el Código 10. Éste sirve de punto de integración entre todos los módulos y sintetiza la funcionalidad del PC (Personal Computer). En detalle el módulo cuenta una función __init__ que inicializa los elementos que componen el pc, y su vez los asocia en dos grupos principales Hardware y Software (módulos del paquete pc). También, cuenta con una función llamada push_start_button que se encarga de encender el equipo y retorna True cuando el proceso se realizó correctamente.

```
from pc.hardware import Hardware, producers
from pc.component import Component
from pc.software import Software, os


class Pc:

    def __init__(self, ram: Component = None, storage: Component = None,
                 network_card: Component = None, processor: Component = None,
                 motherboard: Component = None, os_name: str = None,
os_version: str = None):
        self.hardware = Hardware(ram, storage, network_card, processor,
motherboard)
        self.software = Software(producers[3], os_name, os_version,
self.hardware)

    def push_start_button(self) -> bool:
        try:
            self.hardware.start()
            self.software.start()
            return True
```

```
19.         except ValueError:
20.             return False
21.
22.
```

Código 10. Módulo pc

En conclusión, los objetivos de las pruebas para este paquete serán: verificar el funcionamiento de la clase Pc (prueba de sistema), verificar el funcionamiento en conjunto de los diferentes módulos que conforman el paquete pc, es decir, la integración de Component, Hardware y Software en la clase Pc (pruebas de integración) y finalmente verificar el funcionamiento de cada módulo por separado (pruebas unitarias).

2.6.1.2.1. Pruebas de sistema

Las pruebas de sistema son las más simples de implementar, dado que dependen del correcto funcionamiento del paquete. Éstas se realizan mediante la ejecución directa del paquete o en algunos casos se implementan dentro del archivo de pruebas. Si el sistema retorna un valor incorrecto se da por entendido que existe un error a nivel de sistema.

En el Código 11 se puede apreciar el procedimiento para ejecutar la clase Pc y el resultado al tratar de encender el pc. Dado que el resultado de la función push_start_button es False, podemos identificar que el pc no logro encender. Es decir, nuestro paquete pc funciona incorrectamente y tiene errores a nivel de sistema.

```
>>> import pc.pc as pc
>>> from pc.component import Component
>>> from pc.software import os
>>> from pc.hardware import producers
>>> ram = Component(producers[3], 'RAM DDR4 2333 MHz')
... processor = Component(producers[5], 'PROCESADOR i7 4.7Ghz')
... storage = Component(producers[2], 'DISCO DURO 250GB SPACE')
... network_card = Component(producers[2], 'TARJETA DE RED')
>>> computer= pc.Pc(ram, storage, network_card, processor)
>>> computer.push_start_button()

False
```

Código 11. Ejecución por consola de la clase Pc del módulo pc

Para brindar mayor claridad se migra la prueba de sistema desde ejecución por consola al archivo test_pc.py visualizado en el Figura 16. Sin embargo, se reitera que las pruebas de sistema no necesariamente deben de estar incluidas en el archivo de pruebas.

Al Código 12 se trasladó la prueba de sistema realizada en el Código 11. La nueva prueba usa el ejecutor unittest y nos permite identificar algunos elementos que conforman las clases de pruebas definidas para este ejecutor:

- La clase de prueba siempre debe heredar de la clase TestCase del módulo unittest. Esto garantiza que el intérprete de Python identifique que elementos pertenecen a las pruebas.

- En unittest las funciones por convención comienzan con el prefijo “test_”. Sin embargo, si las nombramos sin el prefijo la prueba sigue ejecutándose, esto se debe a que dicha función sigue siendo un método de la clase de prueba que hereda de TestCase.

- La función __main__ de Python debe ejecutar el comando unittest.main() para que el archivo sea reconocido como un caso de prueba.

```
1.  import unittest
2.  import pc.pc as pc
3.  from pc.component import Component
4.  from pc.hardware import producers
5.
6.
7.  class MyTestCase(unittest.TestCase):
8.      def test_system(self):
9.          ram = Component(producers[3], 'RAM DDR4 2333 MHz')
10.         processor = Component(producers[5], 'PROCESADOR i7 4.7Ghz')
11.         storage = Component(producers[2], 'DISCO DURO 250GB SPACE')
12.         network_card = Component(producers[2], 'TARJETA DE RED')
13.         computer = pc.Pc(ram, storage, network_card, processor)
14.         self.assertTrue(computer.push_start_button())
15.
16.
17. if __name__ == '__main__':
18.     unittest.main()
19.
```

Código 12. Prueba de sistema mediante unittest

Para ejecutar la prueba debe usarse el comando del Código 13. Es de resaltar que los resultados obtenidos se explicaran en secciones posteriores. Sin embargo, es claro que el paquete pc sigue fallando porque no se ha cambiado ninguna línea su lógica.

```
python test_pc.py
```

Código 13. Comando de ejecución para las pruebas

2.6.1.2.2. Pruebas de integración

Las pruebas de integración necesitan de un ejecutor de pruebas para brindar organización y trazabilidad. A diferencia de las pruebas de sistema, éstas generalmente son ejecutadas por un servidor automatización, por lo que siempre deben ir en el archivo de pruebas. Dado el caso de que no exista un servidor de automatización éstas son ejecutadas solo cuando las pruebas de sistema fallan.

Los módulos component, software y hardware permiten visualizar las relaciones de integración que existen en el paquete pc. Mediante estos se puede definir un esquema completo de pruebas de integración. Sin embargo, para sección solo se definirá una prueba de integración que sirva de ejemplo.

Prueba de integración entre la clase Component y clase Hardware: La clase Component se utiliza en múltiples ocasiones como tipo para variables de instancia de la clase Hardware. En consecuencia, se puede definir que existe una dependencia entre las dos clases, la cual podría llegar a ser un punto de fallo en el paquete pc.

En el Código 14 podemos visualizar el método resultante de este caso de prueba. Éste, puede parecer muy similar al método de la prueba de sistema. Sin embargo, para la prueba de sistema existían múltiples módulos interactuando entre sí y para esta prueba solo interactúan dos módulos. Es de resaltar que para los nombres de los casos de prueba de integración se recomienda usar el prefijo “test_int”, aunque su uso es plenamente por organización.

```
1.      def test_int_comp_hard(self):
2.          ram = Component(producers[3], 'RAM DDR4 2333 MHz')
3.          processor = Component(producers[5], 'PROCESADOR i7 4.7Ghz')
4.          storage = Component(producers[2], 'DISCO DURO 250GB SPACE')
5.          network_card = Component(producers[2], 'TARJETA DE RED')
6.          hw = Hardware(ram, storage, network_card, processor)
7.          hw.start()
8.          self.assertTrue(hw.is_up())
```

Código 14. Caso de prueba de integración

Antes de continuar, cabe destacar que se está presentando un problema de repetición de código. Para estos casos, unittest brinda la posibilidad de utilizar el método setup. Este método se ejecuta antes de iniciar las pruebas y permite tener variables de instancia para ser reusadas en diferentes puntos del código de prueba. Sin embargo, cabe aclarar que las variables erróneas definidas en setup, pueden ser un punto de fallo difícil de detectar desde el nivel de integración. En el Código 15 se puede visualizar el resumen de la clase MyTestCase con la implementación del método setup.

```
1.  import unittest
2.  import pc.pc as pc
3.  from pc.component import Component
4.  from pc.hardware import producers, Hardware
5.
6.
7.  class MyTestCase(unittest.TestCase):
```

```
8.      def setUp(self) -> None:
9.          self.ram = Component(producers[3], 'RAM DDR4 2333 MHz')
10.         self.processor = Component(producers[5], 'PROCESADOR i7 4.7Ghz')
11.         self.storage = Component(producers[2], 'DISCO DURO 250GB SPACE')
12.         self.network_card = Component(producers[2], 'TARJETA DE RED')
13.
14.     def test_system(self):
15.         computer = pc.Pc(self.ram, self.storage, self.network_card,
    self.processor)
16.         self.assertTrue(computer.push_start_button())
17.
18.     def test_int_comp_hard(self):
19.         hw = Hardware(self.ram, self.storage, self.network_card,
    self.processor)
20.         hw.start()
21.         self.assertTrue(hw.is_up())
22.
23.
24. if __name__ == '__main__':
25.     unittest.main()
26.
```

Código 15. Refactorización de la clase MyTestCase

Al ejecutar las pruebas de nuevo, podemos verificar que unittest nos sigue indicando que existen errores. Esto indica que el paquete pc tiene errores a nivel de integración, lo cual era de esperarse dado que falló a nivel de sistema.

2.6.1.2.3. Pruebas unitarias

Las pruebas unitarias verifican los puntos internos de sistema. Se consideran las más importantes para cualquier tipo de código, y al igual que las pruebas de integración generalmente son ejecutadas recurrentemente por un servidor de automatización. En las pruebas unitarias se realizan diferentes verificaciones según el tipo de estructura de datos a la cual se le realizará la prueba. Para las clases se prueban todas las variables instancia y los métodos que no impliquen comunicación con otros módulos. Para las funciones individuales se prueban combinaciones de parámetros de entrada y finalmente para elementos que guardan constantes se realizan pruebas sobre los valores esperados.

Para el ejemplo de pruebas unitarias se usarán los módulos hardware y component.

En primer lugar, al analizar el módulo component se puede identificar que cuenta con una sola clase, y ésta a su vez con tres variables de instancia y un método que deben ser probados. Posteriormente se analiza el módulo hardware y resulta que contiene una lista y una clase, la cual a su vez contiene ocho variables de instancia y ningún método para ser probado. Es una

buena práctica probar todos los elementos de una clase para evitar posibles fallos en el funcionamiento. Sin embargo, si deseáramos que nuestras pruebas sean menos extensas podemos implementar solo las variables de instancia y métodos que interactúen con otros módulos.

En el Código 16 podemos identificar el resumen de todas las pruebas. Algunos detalles importantes de dicho código son:

- Los métodos de la clase Hardware no cuentan como pruebas unitarias porque dependen de la clase Component para ejecutarse correctamente. Toda prueba unitaria debe garantizar que la respuesta solo depende del elemento a ser probado. Por ejemplo, en las pruebas de las variables de tipo Component de la clase Hardware (ram, storage, processor, network_card y motherboard) solo es posible verificar el tipo, puesto que de otra manera se rompe el principio de única responsabilidad en la prueba unitaria.
- En el código se pueden identificar cuatro tipos validación: validación de instancia (self.assertIsInstance), validación de igualdad (self.assertEqual), validación de diferencia (self.assertNotEqual) y validación de numérica para el caso de "mayor que" (self.assertGreater).
- Las pruebas unitarias definidas no son las únicas posibles, dado que según la visión del desarrollador sobre el paquete pc pueden resultar otras alternativas.
- En algunos casos es posible utilizar doble validación en una misma prueba unitaria, mientras el elemento a valuar sea el mismo. Por ejemplo, el método *test_un_hard_state* evalúa en dos ocasiones la variable state de formas diferentes.

```
1.  import unittest
2.  import pc.pc as pc
3.  from pc.component import Component
4.  from pc.hardware import producers, Hardware
5.  from typing import List
6.
7.
8.  class MyTestCase(unittest.TestCase):
9.      def setUp(self) -> None:
10.         self.ram = Component(producers[3], 'RAM DDR4 2333 MHz')
11.         self.processor = Component(producers[5], 'PROCESADOR i7 4.7Ghz')
12.         self.storage = Component(producers[2], 'DISCO DURO 250GB SPACE')
13.         self.network_card = Component(producers[2], 'TARJETA DE RED')
14.         self.hw = Hardware(self.ram, self.storage, self.network_card, self.processor)
15.
16.     def test_system(self):
17.         computer = pc.Pc(self.ram, self.storage, self.network_card, self.processor)
18.         self.assertTrue(computer.push_start_button())
19.
20.     def test_int_comp_hard(self):
```

```
21.         hw = Hardware(self.ram, self.storage, self.network_card, self.processor)
22.         hw.start()
23.         self.assertTrue(hw.is_up())
24.
25.     def test_un_comp_name(self):
26.         ssd_name = 'SSD 1TB'
27.         ssd = Component(producers[2], ssd_name)
28.         self.assertEqual(ssd_name, ssd.name)
29.
30.     def test_un_comp_brand(self):
31.         cd = Component(producers[2], 'CD 500Mb')
32.         self.assertEqual(producers[2], cd.brand)
33.
34.     def test_un_comp_str(self):
35.         cd_name = 'CD 500Mb'
36.         cd = Component(producers[2], cd_name)
37.         self.assertEqual(cd.__str__(), f'Yo soy {cd.name} diseñado por {producers[2]}
    ')
38.
39.     def test_un_comp_state(self):
40.         graphic_card = Component(producers[5], 'IRIS GRAPHICS 10000')
41.         graphic_card.state = Hardware.POWER_ON
42.         self.assertEqual(graphic_card.state, Hardware.POWER_ON)
43.
44.     def test_un_hard_state(self):
45.         hw = Hardware(self.ram, self.storage, self.network_card, self.processor)
46.         hw.state = Hardware.POWER_ON
47.         self.assertEqual(hw.state, Hardware.POWER_ON)
48.         hw.state = Hardware.POWER_OFF
49.         self.assertNotEqual(hw.state, Hardware.POWER_ON)
50.
51.     def test_un_hard_comp_a(self):
52.         self.assertIsInstance(self.hw.ram, Component)
53.
54.     def test_un_hard_comp_b(self):
55.         self.assertIsInstance(self.hw.storage, Component)
56.
57.     def test_un_hard_comp_c(self):
58.         self.assertIsInstance(self.hw.network_card, Component)
59.
60.     def test_un_hard_comp_d(self):
61.         self.assertIsInstance(self.hw.processor, Component)
62.
63.     def test_un_hard_comp_e(self):
64.         self.assertIsInstance(self.hw.motherboard, Component)
```

```
65.
66.     def test_un_producers_a(self):
67.         self.assertIsInstance(producers, List)
68.
69.     def test_un_producers_b(self):
70.         self.assertGreater(len(producers), 0)
71.
72.
73. if __name__ == '__main__':
74.     unittest.main()
75.
```

Código 16. Archivo final de pruebas test_pc.py

En siguiente sección se analizarán los resultados obtenidos mediante las pruebas de sistema, de integración y unitarias definidas para el paquete pc.

2.6.1.2.4. Interpretación de resultados

Los resultados obtenidos al ejecutar una prueba son intuitivos y facilitan la tarea de mantener el código. En general, los resultados de las pruebas están conformados por los siguientes elementos:

- Número de pruebas ejecutadas.
- Número de pruebas ejecutadas correctamente.
- Número de pruebas ejecutadas incorrectamente.
- Tiempo de ejecución.
- Trazabilidad de los errores en caso de que existan.
- Descripción rápida del error
- Resultado final de la prueba.

Interpretar los resultados de una prueba pasa por dos etapas. La primera, consisten en identificar la prueba y el motivo por el que dicha prueba falló. La segunda etapa se denomina etapa de conclusiones, depende del nivel de las pruebas que fallaron y la relación entre éstas. Dado que, si las pruebas que fallaron son unitarias, la solución únicamente implica modificar el objeto para obtener el resultado esperado. Por el contrario, si las pruebas que fallaron son de diferentes niveles se debe analizar con mayor cuidado los resultados y tratar de solucionar el error desde el nivel más bajo hasta el nivel más alto.

2.6.1.2.4.1. Etapa de identificación

En el Código 17 podemos identificar todos los elementos que conforman los resultados de una prueba. Para este caso, tenemos que el sistema retorna que se ejecutaron 14 pruebas, de las cuales fallaron 3 y 11 finalizaron correctamente. De las 3 pruebas que fallaron la descripción del error es la siguiente:

1. **AssertionError: None is not an instance of <class 'pc.component.Component'>:** Este error nos indica que la verificación de instancia no es correcta y que el valor de la variable de entrada para el método era None. Entre las pruebas tenemos 6 que utilizan el método assertIsInstance para verificar las variables de instancia de la clase Hardware. Encontrar el error solo con la descripción para este caso no es posible. Sin embargo, al recurrir a la traza del error podemos visualizar que el problema se presentó en la línea 64, es decir, en la prueba test_un_hard_comp_e.
2. **AssertionError: False is not true:** Este error nos indica que la verificación de un valor booleano no fue correcta. Por suerte para este caso, la única prueba que utiliza verificación por booleano es test_system.
3. **ValueError: Hardware doesnt start:** Este error es el más particular de los obtenidos en la lista, puesto que no implica una falla en el sistema validaciones si no que indica que existió una excepción no controlada. Esto puede identificarse en la primera palabra de la descripción corta, dado que si el error proviene de una validación la primera palabra será AssertionError. Mientras que, si el error proviene de una excepción, la primera palabra será el nombre de la dicha excepción.

De las descripciones de los errores cabe resaltar que las pruebas no se ejecutan en orden. Esto se debe a que unittest trata de garantizar la independencia entre pruebas.

2.6.1.2.4.2. Etapa de conclusiones

Para esta etapa tenemos que los resultados del Código 26 muestran errores en todos los niveles de las pruebas. Siguiendo las recomendaciones para identificar errores, primero nos concentramos en el error de la descripción 1, puesto que éste hace referencia a una prueba unitaria. Claramente este error nos indica que la variable que falla se llama motherboard. Con esto en mente, subimos de nivel y verificamos si dicha variable puede estar interfiriendo en el error de las pruebas de integración. Y efectivamente, el error de la prueba de integración se genera por una excepción de tipo AttributeError cuando se trata de asociar un valor a la variable motherboard. Esto indica que el error proviene de la inicialización incorrecta de la clase Hardware.

En conclusión, corregir la prueba requiere de crear un nuevo objeto de la clase Component con los valores de una motherboard y asociarla a la inicialización de la clase Hardware para el método setup y para el método test_int_comp_hard. También se debe asociar dicha variable en la prueba test_system a la clase Pc para que por herencia el error quede solucionado a nivel de sistema.

```
1. ======================================================================
2. ERROR: test_int_comp_hard (__main__.MyTestCase)
3. ----------------------------------------------------------------------
4. Traceback (most recent call last):
5.   File "A:\Archivos\UdeA\Trabajo de grado\unit-test\pc\hardware.py", line 24,
   in start
```

```
6.      self.motherboard.state = self.POWER_ON
7.  AttributeError: 'NoneType' object has no attribute 'state'
8.
9.  During handling of the above exception, another exception occurred:
10.
11. Traceback (most recent call last):
12.   File "test_pc.py", line 22, in test_int_comp_hard
13.     hw.start()
14.   File "A:\Archivos\UdeA\Trabajo de grado\unit-test\pc\hardware.py", line 28,
    in start
15.     raise ValueError('Hardware doesnt start')
16. ValueError: Hardware doesnt start
17.
18. ======================================================================
19. FAIL: test_system (__main__.MyTestCase)
20. ----------------------------------------------------------------------
21. Traceback (most recent call last):
22.   File "test_pc.py", line 18, in test_system
23.     self.assertTrue(computer.push_start_button())
24. AssertionError: False is not true
25.
26. ======================================================================
27. FAIL: test_un_hard_comp_e (__main__.MyTestCase)
28. ----------------------------------------------------------------------
29. Traceback (most recent call last):
30.   File "test_pc.py", line 64, in test_un_hard_comp_e
31.     self.assertIsInstance(self.hw.motherboard, Component)
32. AssertionError: None is not an instance of <class 'pc.component.Component'>
33.
34. ----------------------------------------------------------------------
35. Ran 14 tests in 0.003s
36.
37. FAILED (failures=2, errors=1)
38.
```

Código 17. Resultados de las pruebas del archivo test_pc.py.

Para finalizar esta sección se implementan los cambios concluidos de los resultados del Código 17. La clase corregida retorna la siguiente respuesta (Ver Código 18):

```
1.  Ran 14 tests in 0.004s
2.
3.  OK
4.
```

```
5. Process finished with exit code 0
```

Código 18. Resultados corregidos de las pruebas de test_pc.py

2.6.2. Pruebas de software en Java

Java cuenta con uno de los ecosistemas más grande de pruebas en el mundo de los lenguajes de programación. En Java los entornos de pruebas pasaron de ser simples librerías para convertirse en frameworks especializados en todo tipo de pruebas.

Al igual que en Python, la estructura de las pruebas en Java también puede resumirse en dos preguntas ¿Cómo probar código? Y ¿Cómo implementar código? En la Figura 18 resumimos los temas resultantes de dichas preguntas mediante un mapa conceptual.

Cabe mencionar que las siguientes secciones solo son una pequeña parte de las bondades que brindan los framework de pruebas en Java.

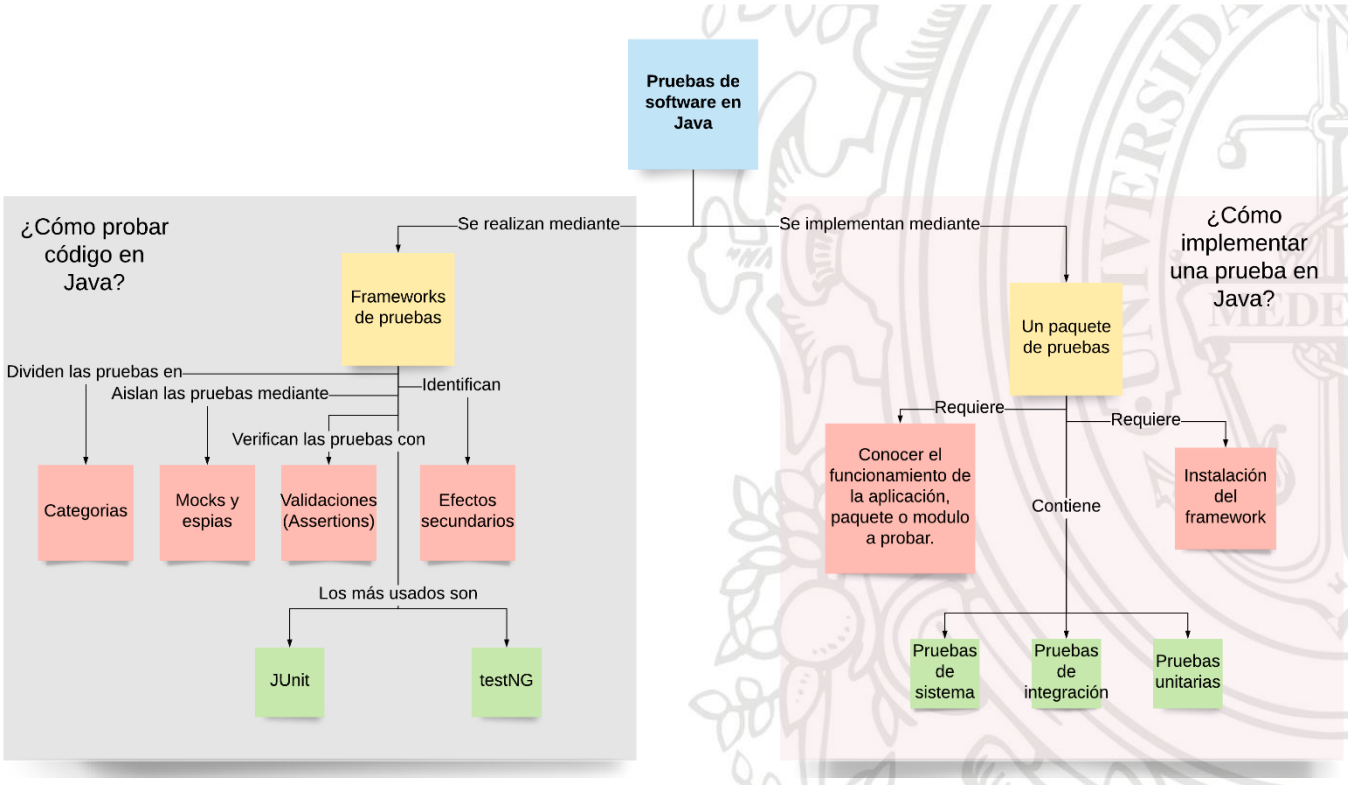

Figura 18. Mapa conceptual que resume pruebas de software en Java

2.6.2.1. ¿Cómo pensar para probar código en Java?

En Java para realizar pruebas se utilizan frameworks. Suele confundirse el termino framework con librería. Sin embargo, son dos elementos diferentes porque un framework generalmente se encuentra formado por un super compilado de librerías que a su vez proporciona un marco de

trabajo de común. En resumen, un framework se deferencia de una librería o de un conjunto de éstas porque define una normativa clara para crear código. Algunos de las principales normas que se pueden identificar en un framework son:

- Define una estructura de directorios.
- Proporcional un conjunto de anotaciones comunes.
- Define una estructura de ejecución.
- Exige aislamiento de sus paquetes.

En Java existen muchos frameworks de desarrollo, pero la mayor parte de los usuarios se concentra en los dos más usados, Junit y TestNG. Sin embargo, la diferencia de uso entre estos dos es realmente significativa mientras Junit (versión 4 y 5) es usada por más de 1.4 millones de repositorios en GitHub, TestNG únicamente es usada por 156,463 repositorios. Es decir, junit se usa aproximadamente diez veces más que TestNG en GitHub.

En el Cuadro 5 se puede visualizar una comparativa entre JUnit y TestNG.

Característica	**JUnit 5**	**TestNG**	**Conclusión**
Anotaciones	Si. Basado en anotaciones.	Si. Basado en anotaciones.	Ambos frameworks basan su sistema de pruebas en cómodas anotaciones.
Facilidad de uso	Esta formado por varios módulos. Algunos de estos son: - JUnit Jupiter - JUnit Platform - JUnit parameters Entre otros.	Presenta un único modulo que facilita su importación.	TestNG es mejor en términos de uso al estar formado por único uso
Soporte para IDE	Excelente	Bueno	Gana JUnit puesto que extiende su soporte a editores de texto como VSCode
Proveedores de datos	Excelente. Soporta Enum, CSV, XLSX, JSON, clases, etc..	Malo. Soporte xml y clases.	Gana JUnit porque permite ingresar datos a las pruebas desde una mayor cantidad de fuentes de datos

Conjuntos de pruebas	Soporta conjuntos de pruebas	Soporta conjuntos de prueba	Ambos soportan conjuntos de pruebas, es decir, ejecución de paquetes específicos de pruebas.
Reportes HTML	Si, reportes con flexibles HTML5	Si, reportes básicos.	Gana JUnit porque brinda reportes flexibles muchos más atractivos a la vista. Los reportes de TestNG son obsoletos.
Validaciones	Si	Si	Ambas cuentan con un robusto sistema de validaciones.
Ejecución paralela	No	Si	Gana TestNG pues la ejecución de pruebas puede darse de manera paralela.

Cuadro 5. Comparativa entre JUnit 5 y TestNG

Como se pudo visualizar en la comparativa anterior el líder indiscutible en el entorno de pruebas Java es Junit, por ende, se utilizará para realizar las pruebas en la siguiente sección.

2.6.2.1.1. Validaciones

En la Java el sistema de validaciones cumple la misma función que en Python. Es decir, verificar que las condiciones determinadas en las pruebas se cumplan correctamente. Tanto para TestNG como para JUnit el sistema de validación contiene las mismas estructuras.

En el Cuadro 6 se presentará una comparativa entre las funciones de validación para cada framework con su respectivo equivalente.

JUnit 5	TestNG	Equivalente
Assertions.assertTrue(a)	Assert.assertTrue(a)	if (¡a) throw Excepction
Assertions.assertFalse(a)	Assert.assertFalse(a)	if (a) throw Excepction
Assertions.assertEquals(a, b)	Assert.assertEquals(a, b)	if (a != b) throw Excepction

Assertions.assertArrayEquals([a], [b])	Assert.assertEquals([a], [b])	for (int i=0; i<a.lenght; i++)
		if (a[i] != b[i])
		throw Exception

Assertions.assertSame(a, b)	Assert.assertSame(a, b)	if (a == b) return;

		throw Exeption
Assertions.assertNull(a)	Assert.assertSame(a)	if (a != null) throw Exception
Assertions.assertThrows(Exc.class, Func)	No aplica	try{ Func(); throw Exception;} catch (Exc.class){return;}

Cuadro 6. Comparativa de validaciones entre JUnit 5 y TestNG

2.6.2.1.2. Mock y espías

Unas de las características que hace los frameworks para pruebas en Java tan potentes, es la capacidad de encapsulamiento. Es decir, la capacidad de simular elementos externos a la prueba para garantizar que solo se está evaluando un elemento. Mientras en Python era obligatorio crear elementos controlados para luego usarlos en las pruebas, en Java los Mock y espias garantizan que dichos elementos externos retornen una respuesta encapsulada de los datos. La librería que usan los dos frameworks para este propósito es la misma, Mockito.

La diferencia entre los Mocks y espías consiste es que los Mocks solo son interfaces huecas que no almacenan valores. Mientras los espías son objetos que simulan ser una clase.

2.6.2.1.3. Catalogación

En Java los sistemas de catalogación permiten asociar un TAG a una prueba, de manera que sea posible ejecutar solo conjuntos específicos. Los sistemas de catalogación permiten incluso ejecutar pruebas que contengan una clase especifica o parametrizar ciertos elementos para que modifiquen la respuesta de múltiples pruebas.

Es de resaltar que los Mocks, espías y sistemas de catalogación serán más claros en la siguiente sección donde se aborden ejemplos de los mismos.

Algunas buenas prácticas para comenzar con las pruebas en Java son:

1. **Usar un framework para pruebas:** Los frameworks de pruebas facilitan la configuración y ejecución de las pruebas. También permite: soporte de anotaciones, creación de ciclos específicos de prueba, omisión de casos de prueba, soporte para parámetros por ambiente y automatización de pruebas ante compilación.
2. **Usar la metodología TDD (Test-driven developmen):** El desarrollo basado en pruebas (TDD) es un proceso de desarrollo de software en el que las pruebas se escriben en función de los requisitos antes de que comience la codificación. El objetivo es escribir pruebas que cubran todos los requisitos. TDD es excelente ya que conduce a un código modular simple que es fácil de mantener.
3. **Cobertura de código:** El código cubierto por las pruebas debe ser el mayor posible. Sin embargo, las pruebas no deben caer en ejecuciones innecesarias. Para identificar el código cubierto por las pruebas unitarias se necesita usar herramientas como: Clover, Cobertura, JaCoCo o SonarQube.

4. **Ejecute test contenidos:** No se debe depender de clases externas para probar el código. Por el contrario, se debe realizar toda la configuración necesaria para que la prueba funcione dentro de la clase.
5. **Uso de asserts:** El objetivo de las pruebas es determinar por si mismos si el código cumple con los requisitos propuestos en el desarrollo de la aplicación. De tal manera que las pruebas sean mantenibles y claras. Es de resaltar que la prueba no debe usar impresiones por consola para identificar si se cumplen o no requisitos. En cambio, se recomienda usar las herramientas de verificaciones proporcionadas por el framework, de tal manera que a largo plazo sea posible automatizar dichas pruebas.
6. **Cree pruebas determinísticas:** El objetivo de las pruebas debe permitir mantener el código a largo plazo. Es decir, el resultado de las pruebas no debe variar con el tiempo.

2.6.2.2. ¿Cuál es la estructura de una prueba en Java?

Para brindar mayor claridad acerca de cómo se realizan las pruebas en Java, se implementará el mismo ejemplo utilizado en Python (Sección 2.6.1.2).

Inicialmente, al igual que se planteó en Python, el primer paso para realizar la estructuración de las pruebas es reconocer el proyecto en el cual se va a trabajar.

En la Figura 19 se puede visualizar la estructura de directorios que maneja el proyecto. Mediante ésta se puede notar que existe un paquete principal llamado pc, el cual se nombra con la siguiente convención: "url-inversa.paquete". Es decir, el nombre completo del paquete es **co.edu.udea.gita.pc .** El nombre del paquete aplica tanto para el código principal como para la carpeta de pruebas. Cabe resaltar que Java usa paquetes inclusive para gestionar sus pruebas mientras que en Python el archivo de pruebas no se puede considerar como un paquete.

Otro elemento importante en la Figura 20 es el archivo *pom.xml*. Este permite identificar que el proyecto sigue una estructura tipo Maven. En el Código 28 se puede visualizar el contenido del archivo, el cual nos revela que para este caso solo se usa el sistema de instalación de dependencias. Las dependencias instaladas mediante Maven son JUnit 5.4.2 y Mockito 2.21.0, con un alcance (scope) test. Es decir, las librerías no se incluirán en una posible compilación, solo se descargan y ejecutan cuando se invoca al framework de pruebas.

En el Código 19 también se nota que para el proyecto usa Java 1.8, esto se puede apreciar en las etiquetas *source* y *target* del plugin de compilación. Este plugin viene por defecto para poder generar los compilados necesarios que ejecutan la aplicación Java.

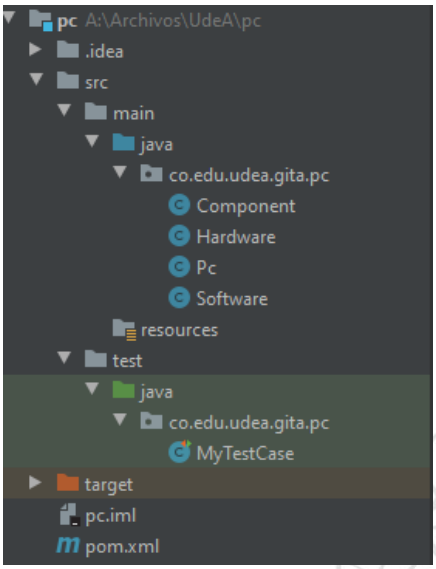

Figura 19. Estructura de directorios para Java

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <project xmlns="http://maven.apache.org/POM/4.0.0"
3.           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
5.      <modelVersion>4.0.0</modelVersion>
6.
7.      <groupId>co.edu.udea.gita</groupId>
8.      <artifactId>pc</artifactId>
9.      <version>1.0-SNAPSHOT</version>
10.     <build>
11.         <plugins>
12.             <plugin>
13.                 <groupId>org.apache.maven.plugins</groupId>
14.                 <artifactId>maven-compiler-plugin</artifactId>
15.                 <configuration>
16.                     <source>8</source>
17.                     <target>8</target>
18.                 </configuration>
19.             </plugin>
20.         </plugins>
21.     </build>
22.
23.     <properties>
24.         <maven.compiler.target>1.8</maven.compiler.target>
25.         <maven.compiler.source>1.8</maven.compiler.source>
26.     </properties>
27.
```

```
    <dependencies>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-api</artifactId>
            <version>5.4.2</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-engine</artifactId>
            <version>5.4.2</version>
            <scope>test</scope>
        </dependency>
        <dependency>
           <groupId>org.mockito</groupId>
           <artifactId>mockito-core</artifactId>
           <version>2.21.0</version>
           <scope>test</scope>
       </dependency>
    </dependencies>
    <dependency>
            <groupId>org.mockito</groupId>
            <artifactId>mockito-junit-jupiter</artifactId>
            <version>2.21.0</version>
            <scope>test</scope>
        </dependency>

</project>
```

Código 19. Archivo pom.xml del proyecto pc

En Java la ubicación de las pruebas siempre debe realizarse en el directorio test de la estructura Maven, dentro de un paquete con el mismo nombre del paquete al cual se le realizarán las pruebas. No existe una convención generalizada respecto al nombramiento de las clases de prueba. Dado que JUnit solo utiliza como referencia el directorio de pruebas para identificar cuáles son las pruebas para ejecutar. Sin embargo, si existen algunas convenciones para nombrar los métodos de pruebas. Las dos más importantes son:

1. **NombreDelMetodo_EstadoASerProbado_ComportamientoEsperado:** Hace años era una de las convenciones más usadas, pero debido a las contantes criticas por la confusión que se generaba cuando era necesario refactorizar métodos en el código principal quedó en desuso. Un ejemplo es: isAdult_AgeLessThan18_False
2. **test[Funcionalidad]:** Su principal ventaja es la facilidad de lectura cuando se está desarrollando. Sin embargo, el prefijo test en Java se considera redundante y en algunos casos prefiere dejarse de lado. Un ejemplo es: testIsNotAnAdultIfAgeLessThan18.

A continuación, en los siguientes códigos (Código 20, Código 21, Código 22 y Código 23) se ilustran las estructuras resultantes de la implementación del proyecto pc en Java.

Algunos detalles para resaltar provenientes de comparar el código obtenido en Java respecto al código obtenido en Python son:

- Tanto Python como Java manejan sistemas basados en paquetes. Sin embargo, para Python los archivos se consideran módulos y para Java son contenidos únicos de clases. Es decir, cada archivo solo puede exponer una única clase. Esto implica que en Python sea posible agregar múltiples estructuras a los archivos y en Java no.
- En Java la privacidad es un elemento fundamental, por lo que existe control de acceso a los elementos de cada clase. Mientras que en Python la privacidad no se tiene en cuenta de manera nativa.
- La importación de dependencias en Java es más sencilla y eficiente.
- El código Java es mucho más extenso que el código Python.
- Las variables estáticas que se usaban en Python, en Java se convierten en constantes con la palabra final en su declaración. Es decir, constantes únicas que no pueden modificarse en tiempo de ejecución.
- Las listas que se encontraban fuera de la clase principal en Python, en Java se reubicaron dentro de la clase como una constante.
- En Java al inicio del archivo existe una instrucción que indica que todas las clases que se encuentren en el mismo nivel en el paquete indicado pueden ser cargadas sin necesidad de ser importadas “package co.edu.udea.gita.pc;”.

```
1.  package co.edu.udea.gita.pc;
2.
3.  public class Component {
4.
5.      private String brand;
6.      private String name;
7.      private String state;
8.
9.      public Component(String brand, String name) {
10.         this.brand = brand;
11.         this.name = name;
12.     }
13.
14.     @Override
15.     public String toString() {
16.         return "Yo soy " + name + ", diseñado por" + brand + ' ';
17.     }
18.
19.     public String getBrand() {
20.         return brand;
21.     }
```

```
22.
23.     public void setBrand(String brand) {
24.         this.brand = brand;
25.     }
26.
27.     public String getName() {
28.         return name;
29.     }
30.
31.     public void setName(String name) {
32.         this.name = name;
33.     }
34.
35.     public String getState() {
36.         return state;
37.     }
38.
39.     public void setState(String state) {
40.         this.state = state;
41.     }
42. }
43.
```

Código 20. Clase Component del paquete co.edu.udea.gita.pc

```
1.  package co.edu.udea.gita.pc;
2.
3.  import java.util.ArrayList;
4.  import java.util.Arrays;
5.  import java.util.List;
6.
7.  public class Hardware {
8.
9.      public static final List<String> producers = new ArrayList<String>(
10.             Arrays.asList("APPLE", "ASUS", "SAMSUNG", "MICROSOFT", "IBM",
    "INTEL")
11.     );
12.
13.     private static final String POWER_ON = "POWER ON";
14.     private static final String POWER_OFF = "POWER OFF";
15.
16.     private Component ram;
17.     private Component storage;
18.     private Component network_card;
19.     private Component processor;
20.     private Component motherboard;
```

```
21.     private String state;
22.
23.     public Hardware(Component ram, Component storage,
24.                     Component network_card, Component processor,
25.                     Component motherboard) {
26.         this.state = POWER_OFF;
27.         this.ram = ram;
28.         this.storage = storage;
29.         this.network_card = network_card;
30.         this.processor = processor;
31.         this.motherboard = motherboard;
32.     }
33.
34.     public void start() throws NullPointerException {
35.         this.ram.setState(POWER_ON);
36.         this.storage.setState(POWER_ON);
37.         this.network_card.setState(POWER_ON);
38.         this.processor.setState(POWER_ON);
39.         this.motherboard.setState(POWER_ON);
40.         this.state = POWER_ON;
41.     }
42.
43.     public boolean is_up() {
44.         return this.state.equals(POWER_ON);
45.     }
46.
47.     public Component getRam() {
48.         return ram;
49.     }
50.
51.     public void setRam(Component ram) {
52.         this.ram = ram;
53.     }
54.
55.     public Component getStorage() {
56.         return storage;
57.     }
58.
59.     public void setStorage(Component storage) {
60.         this.storage = storage;
61.     }
62.
63.     public Component getNetwork_card() {
64.         return network_card;
65.     }
```

```
66.
67.     public void setNetwork_card(Component network_card) {
68.         this.network_card = network_card;
69.     }
70.
71.     public Component getProcessor() {
72.         return processor;
73.     }
74.
75.     public void setProcessor(Component processor) {
76.         this.processor = processor;
77.     }
78.
79.     public Component getMotherboard() {
80.         return motherboard;
81.     }
82.
83.     public void setMotherboard(Component motherboard) {
84.         this.motherboard = motherboard;
85.     }
86.
87.     public String getState() {
88.         return state;
89.     }
90.
91.     public void setState(String state) {
92.         this.state = state;
93.     }
94. }
95.
```

Código 21. Clase Hardware del paquete co.edu.udea.gita.pc

```
1.  package co.edu.udea.gita.pc;
2.
3.  import java.util.ArrayList;
4.  import java.util.Arrays;
5.  import java.util.List;
6.  import java.util.Objects;
7.
8.  public class Software {
9.
10.     public static final List<String> os = new
    ArrayList<String>(Arrays.asList("WINDOWS", "MAC", "LINUX"));
11.
12.     private static final String ON = "STARTED";
```

```
13.     private static final String OFF = "SHUTDOWN";
14.     private static final String ASLEEP = "ASLEEP";
15.
16.     private String brand;
17.     private String name;
18.     private String version;
19.     private String state;
20.     private Hardware hardware;
21.
22.     public Software(String brand, String name, String version, Hardware
    hardware) {
23.         this.brand = brand;
24.         this.name = name;
25.         this.version = version;
26.         this.hardware = hardware;
27.         this.state = OFF;
28.         Objects.requireNonNull(this.hardware);
29.     }
30.
31.     public void start() {
32.         if (this.hardware.is_up()) {
33.             this.setState(ON);
34.         } else {
35.             throw new NullPointerException("Software cant start without
    hardware");
36.         }
37.     }
38.
39.     public String getBrand() {
40.         return brand;
41.     }
42.
43.     public void setBrand(String brand) {
44.         this.brand = brand;
45.     }
46.
47.     public String getName() {
48.         return name;
49.     }
50.
51.     public void setName(String name) {
52.         this.name = name;
53.     }
54.
55.     public String getVersion() {
```

```
56.         return version;
57.     }
58.
59.     public void setVersion(String version) {
60.         this.version = version;
61.     }
62.
63.     public Hardware getHardware() {
64.         return hardware;
65.     }
66.
67.     public void setHardware(Hardware hardware) {
68.         this.hardware = hardware;
69.     }
70.
71.     public String getState() {
72.         return state;
73.     }
74.
75.     public void setState(String state) {
76.         this.state = state;
77.     }
78. }
79.
```

Código 22. Clase Software del paquete co.edu.udea.gita.pc

```
1. package co.edu.udea.gita.pc;
2.
3.
4. public class Pc {
5.     private Hardware hardware;
6.     private Software software;
7.
8.     public Pc(Component ram, Component storage, Component network_card,
   Component processor,
9.               Component motherboard, String os_name, String os_version) {
10.         this.hardware = new Hardware(ram, storage, network_card, processor,
   motherboard);
11.         this.software = new Software(Hardware.producers.get(3), os_name,
   os_version, hardware);
12.     }
13.
14.     public boolean push_start_button() {
15.         try {
16.             this.hardware.start();
```

```
17.             this.software.start();
18.             return true;
19.         } catch (Exception ex) {
20.             return false;
21.         }
22.     }
23.
24.     public Hardware getHardware() {
25.         return hardware;
26.     }
27.
28.     public void setHardware(Hardware hardware) {
29.         this.hardware = hardware;
30.     }
31.
32.     public Software getSoftware() {
33.         return software;
34.     }
35.
36.     public void setSoftware(Software software) {
37.         this.software = software;
38.     }
39. }
40.
```

Código 23. Clase Pc del paquete co.edu.udea.gita.pc

2.6.2.2.1. Pruebas de sistema, de integración y unitarias.

Dado que los códigos ya se encuentran definidos en Java, se comenzarán a estructurar las pruebas. Para esta sección no explicará como identificar las pruebas a realizar, puesto que en la sección de Python ya explica en detalle que elementos deben ser probados.

En esta sección se explicarán los elementos que conforman una prueba en Java y se modificará el ejemplo de Python para usar Java con JUnit.

Como se explicó en la sección anterior, en Java las pruebas están ubicadas en el directorio test. A su vez, en este directorio se ubican paquetes que contienen las clases con las pruebas. Estas clases contienen los métodos que JUnit usará para identificar los casos de prueba. Cabe resaltar que los métodos ubicados en las clases de prueba tienen una anotación llamada @Test que le permite a JUnit saber que hacen parte de los casos de prueba. Otras anotaciones importantes para JUnit son:

- **@Disabled:** Es usada para deshabilitar un caso o una clase de prueba. Su análogo en JUnit 4 es @Ignore. Esta anotación se usa cuando se tienen casos de prueba en construcción.

- **@ParameterizedTest:** Esta anotación hace posible ejecutar múltiples veces una prueba con diferentes argumentos. Cuando se usa es necesario eliminar la anotación @Test.
- **@RepearedTest:** Esta anotación permite ejecutar pruebas múltiples veces asignando un valor de repetición. La anotación permite conocer la prueba actual y la cantidad total. Al igual que ParameterizedTest debe eliminarse la anotación @Test.
- **@DisplayName:** Permite personalizar el nombre de las pruebas cuando son ejecutadas con IDE. Este nombre es que aparece en el reporte HTML que permite generar JUnit.
- **@BeforeEach:** Esta anotación indica que el método que le lleva debe ser ejecutado antes de cada método de prueba. Su análogo en JUnit 4 es @Before.
- **@AfterEach:** Esta anotación se usa para ejecutar un método justo después de cada método de prueba. Se usa generalmente para resetear alguna propiedad involucrada en la prueba. Su análogo en JUnit 4 es @After
- **@BeforeAll:** Indica al JUnit que debe ejecutar el método antes de todos los métodos de prueba. Su análogo en JUnit 4 es @BeforeClass.
- **@AfterAll:** Esta anotación se usa para ejecutar un método después de todos los métodos de prueba. Su análogo en JUnit 4 es @AfterClass
- **@Tag:** Está anotación se usa para filtrar pruebas. Aplica para clases y métodos.
- **@ExtendedWith:** Anotación para agregar funcionalidades al paquete principal de JUnit.

Anotaciones y clases importantes de Mockito

- **MockitoExtension.class:** Clase para extender la funcionalidad de JUnit mediante las funciones de Mokito.
- **@Mock**: Anotación que se coloca antes de la declaración de un objeto para indicar que al sistema de Mocks que debe tratarse como ficticio. Es decir, debe retornar solo los valores que se le indiquen las funciones de configuración.
- **Mockito.lenient().when(*Mock.function*).thenReturn(*value*):** Este comando permite definir el valor que será retornado (value) ante la solicitud de una función proveniente de un Mock.

Como se puede visualizar en el Código 24, las pruebas en Java son mucho más dinámicas y coherentes. En especial, dichas pruebas sobresalen por su organización, rapidez, flexibilidad, funcionalidad y aislamiento.

Basados en el Código 24 y los elementos anteriores podemos concluir:

- En Java existe la posibilidad de generar objetos ficticios llamados Mock que facilitan el trabajo de las pruebas unitarias y de integración. Dado que garantizan que el código usado no dependa de elementos ajenos a la prueba. Por ejemplo, en el Código 33 los Mocks de la clase Component ejecutan la función setState y aunque en realidad el estado no se mantiene, garantizan la correcta ejecución del código para las pruebas unitarias de la clase Hardware.
- JUnit presenta la posibilidad de extender sus capacidades mediante código de terceros.

- La anotación Tag es de especial utilidad para organizar las pruebas y realizar ejecuciones controladas. Por ejemplo, en el Código 33 la anotación sirvió para permitir distinguir las pruebas unitarias, de integración y de sistema.
- Java con JUnit cuenta con menor cantidad de funciones por defecto que Python con unittest para las validaciones. Sin embargo, mediante librerías de terceros es posible suplir dicha limitación.
- En Java el tipado fuerte brinda una primera capa de seguridad a la aplicación. Dado que evita errores de inicialización como el que se presento para el ejemplo con Python.
- Dado que JUnit funciona mediante un sistema de etiquetas que permite que el nombramiento de los métodos sea dinámico y coherente. Sobre todo comparado con lenguajes como Python donde los nombres de las pruebas pueden ser demasiado extensos.

```
1. package co.edu.udea.gita.pc;
2.
3. import org.junit.jupiter.api.BeforeEach;
4. import org.junit.jupiter.api.Tag;
5. import org.junit.jupiter.api.Test;
6. import org.junit.jupiter.api.extension.ExtendWith;
7. import org.mockito.Mock;
8. import org.mockito.junit.jupiter.MockitoExtension;
9.
10.import static org.junit.jupiter.api.Assertions.*;
11.
12.
13.@ExtendWith(MockitoExtension.class)
14.public class MyTestCase {
15.
16.    @Mock
17.    Component ram;
18.
19.    @Mock
20.    Component processor;
21.
22.    @Mock
23.    Component storage;
24.
25.    @Mock
26.    Component networkCard;
27.
28.    @Mock
29.    Component motherboard;
30.
31.    Component commonComponent;
32.
```

```
33.     Hardware hw;
34.
35.     @BeforeEach
36.     void init(){
37.         this.commonComponent = new Component(Hardware.producers.get(2),
    "common");
38.         this.commonComponent.setState(Hardware.POWER_OFF);
39.         this.hw = new Hardware(this.ram, this.storage, this.networkCard,
    this.processor, this.motherboard);
40.
41.     }
42.
43.     @Test
44.     @Tag("systemTest")
45.     void systemVerification() {
46.         Component ram = new Component(Hardware.producers.get(3), "RAM DDR4
    2333 MHz");
47.         Component processor = new Component(Hardware.producers.get(5),
    "'PROCESADOR i7 4.7Ghz");
48.         Component storage = new Component(Hardware.producers.get(2), "DISCO
    DURO 250GB SPACE'");
49.         Component network_card = new Component(Hardware.producers.get(2),
    "TARJETA DE RED");
50.         Component motherboard = new Component(Hardware.producers.get(1), "RAM
    DDR4 2333 MHz");
51.         Pc pc = new Pc(ram, storage, networkCard, processor, motherboard,
    Software.os.get(0), "XP");
52.         assertTrue(pc.push_start_button());
53.     }
54.
55.     @Test
56.     @Tag("integrationTest")
57.     void startHardware() {
58.         Hardware hw = new Hardware(this.ram, this.storage, this.network_card,
    this.processor, this.motherboard);
59.         hw.start();
60.         assertTrue(hw.is_up());
61.     }
62.
63.     @Test
64.     @Tag("unitTest")
65.     void nameComponentVerification() {
66.         assertEquals(this.commonComponent.getName(), "common");
67.     }
68.
```

```
69.     @Test
70.     @Tag("unitTest")
71.     void brandComponentVerification() {
72.         assertEquals(this.commonComponent.getBrand(),
    Hardware.producers.get(2));
73.     }
74.
75.     @Test
76.     @Tag("unitTest")
77.     void displayComponentVerification() {
78.         assertEquals(this.commonComponent.toString(),
79.                 "Yo soy " + this.commonComponent.getName() + ", diseñado por"
80.                         + this.commonComponent.getBrand() + ' ');
81.     }
82.
83.     @Test
84.     @Tag("unitTest")
85.     void stateComponentVerification() {
86.         assertEquals(this.commonComponent.getState(), Hardware.POWER_OFF);
87.     }
88.
89.     @Test
90.     @Tag("unitTest")
91.     void hardwareStateVerification() {
92.         hw.setState(Hardware.POWER_ON);
93.         assertEquals(hw.getState(), Hardware.POWER_ON);
94.     }
95.
96.     @Test
97.     @Tag("unitTest")
98.     void hardwareRamVerification() {
99.         assertNotNull(hw.getRam());
100.        }
101.
102.        @Test
103.        @Tag("unitTest")
104.        void hardwareStorageVerification() {
105.            assertNotNull(hw.getStorage());
106.        }
107.
108.        @Test
109.        @Tag("unitTest")
110.        void hardwareNetworkVerification() {
111.            assertNotNull(hw.getNetworkCard());
112.        }
```

```
113.
114.          @Test
115.          @Tag("unitTest")
116.          void hardwareProcessorVerification() {
117.              assertNotNull(hw.getProcessor());
118.          }
119.
120.          @Test
121.          @Tag("unitTest")
122.          void hardwareMotherboardVerification() {
123.              assertNotNull(hw.getMotherboard());
124.          }
125.
126.          @Test
127.          @Tag("unitTest")
128.          void hardwareProducersVerification() {
129.              assertNotNull(Hardware.producers);
130.          }
131.
132.          @Test
133.          @Tag("unitTest")
134.          void hardwareProducersLengthVerification() {
135.              assertEquals(Hardware.producers.size(), 6);
136.          }
137.
138.      }
```

Código 24. Prueba con JUnit del paquete co.edu.udea.gita.pc

2.6.2.2.2. Interpretación de resultados

Los resultados de las pruebas en casi la mayoría de los lenguajes de programación comparten una estructura común. Es decir, están diseñados para comprenderse, aunque no se conozca los detalles del código.

Para ejecutar las pruebas mediante Maven por consola es necesario usar el comando ***mvn clean test***. Una vez el comando sea ejecutado se puede visualizar la respuesta del Código 25, donde desde la línea 39 hasta la línea 48 el sistema imprime un resumen de los datos obtenidos en la prueba:

- Cantidad de pruebas ejecutadas.
- Cantidad de pruebas ejecutadas correctamente.
- Cantidad de pruebas ejecutadas incorrectamente.
- Cantidad de pruebas con errores.
- Tiempo de ejecución.

Cabe resaltar que la información de la línea 42 no se repite en la línea 46. Lo que sucede es que JUnit permite identificar la cantidad de pruebas ejecutadas por clase. Es decir, permite visualizar un resumen parcial de las pruebas por clase y un resumen total de las pruebas ejecutadas.

```
1. [INFO] Scanning for projects...
2. [WARNING]
3. [WARNING] Some problems were encountered while building the effective model
   for co.edu.udea.gita:pc:jar:1.0-SNAPSHOT
4. [WARNING] 'build.plugins.plugin.version' for org.apache.maven.plugins:maven-
   compiler-plugin is missing. @ line 12, column 21
5. [WARNING]
6. [WARNING] It is highly recommended to fix these problems because they
   threaten the stability of your build.
7. [WARNING]
8. [WARNING] For this reason, future Maven versions might no longer support
   building such malformed projects.
9. [WARNING]
10. [INFO]
11. [INFO] ------------------------< co.edu.udea.gita:pc >-----------------------
    --
12. [INFO] Building pc 1.0-SNAPSHOT
13. [INFO] --------------------------------[ jar ]---------------------------------
    --
14. [INFO]
15. [INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ pc ---
16. [INFO] Deleting A:\Archivos\UdeA\pc\target
17. [INFO]
18. [INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ pc ---
19. [WARNING] Using platform encoding (Cp1252 actually) to copy filtered
    resources, i.e. build is platform dependent!
20. [INFO] Copying 0 resource
21. [INFO]
22. [INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ pc ---
23. [INFO] Changes detected - recompiling the module!
24. [WARNING] File encoding has not been set, using platform encoding Cp1252,
    i.e. build is platform dependent!
25. [INFO] Compiling 4 source files to A:\Archivos\UdeA\pc\target\classes
26. [INFO]
27. [INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @
    pc ---
28. [WARNING] Using platform encoding (Cp1252 actually) to copy filtered
    resources, i.e. build is platform dependent!
29. [INFO] skip non existing resourceDirectory
    A:\Archivos\UdeA\pc\src\test\resources
```

```
30. [INFO]
31. [INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ pc -
    --
32. [INFO] Changes detected - recompiling the module!
33. [WARNING] File encoding has not been set, using platform encoding Cp1252,
    i.e. build is platform dependent!
34. [INFO] Compiling 1 source file to A:\Archivos\UdeA\pc\target\test-classes
35. [INFO]
36. [INFO] --- maven-surefire-plugin:2.22.0:test (default-test) @ pc ---
37. [INFO]
38. [INFO] -------------------------------------------------------
39. [INFO]  T E S T S
40. [INFO] -------------------------------------------------------
41. [INFO] Running co.edu.udea.gita.pc.MyTestCase
42. [INFO] Tests run: 14, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.274
    s - in co.edu.udea.gita.pc.MyTestCase
43. [INFO]
44. [INFO] Results:
45. [INFO]
46. [INFO] Tests run: 14, Failures: 0, Errors: 0, Skipped: 0
47. [INFO]
48. [INFO] ------------------------------------------------------------------------
    --
49. [INFO] BUILD SUCCESS
50. [INFO] ------------------------------------------------------------------------
    --
51. [INFO] Total time:  2.904 s
52. [INFO] Finished at: 2020-04-12T22:44:31-05:00
53. [INFO] ------------------------------------------------------------------------
    --
54.
```

Código 25. Resultados de pruebas mediante JUnit

Aunque las pruebas mediante Maven son bastante útiles, la manera más productiva de realizar las pruebas en Java es mediante un IDE de código. Cuando se utilizan este tipo de herramientas, podemos acceder a funcionalidades como: ejecución de pruebas parciales de manera inmediata, verificación de porcentaje de código cubierto por las pruebas, control de trazabilidad de errores e incluso debug de las pruebas. Todos estos beneficios generalmente están incluidos de manera nativa e implican un solo click para su ejecución.

3. Resultados

En este capítulo se resumen los resultados derivados de este trabajo de grado. A continuación, en la Figura 20, se identifican las cuatro áreas de impacto de la tesis, así como sus correspondientes módulos.

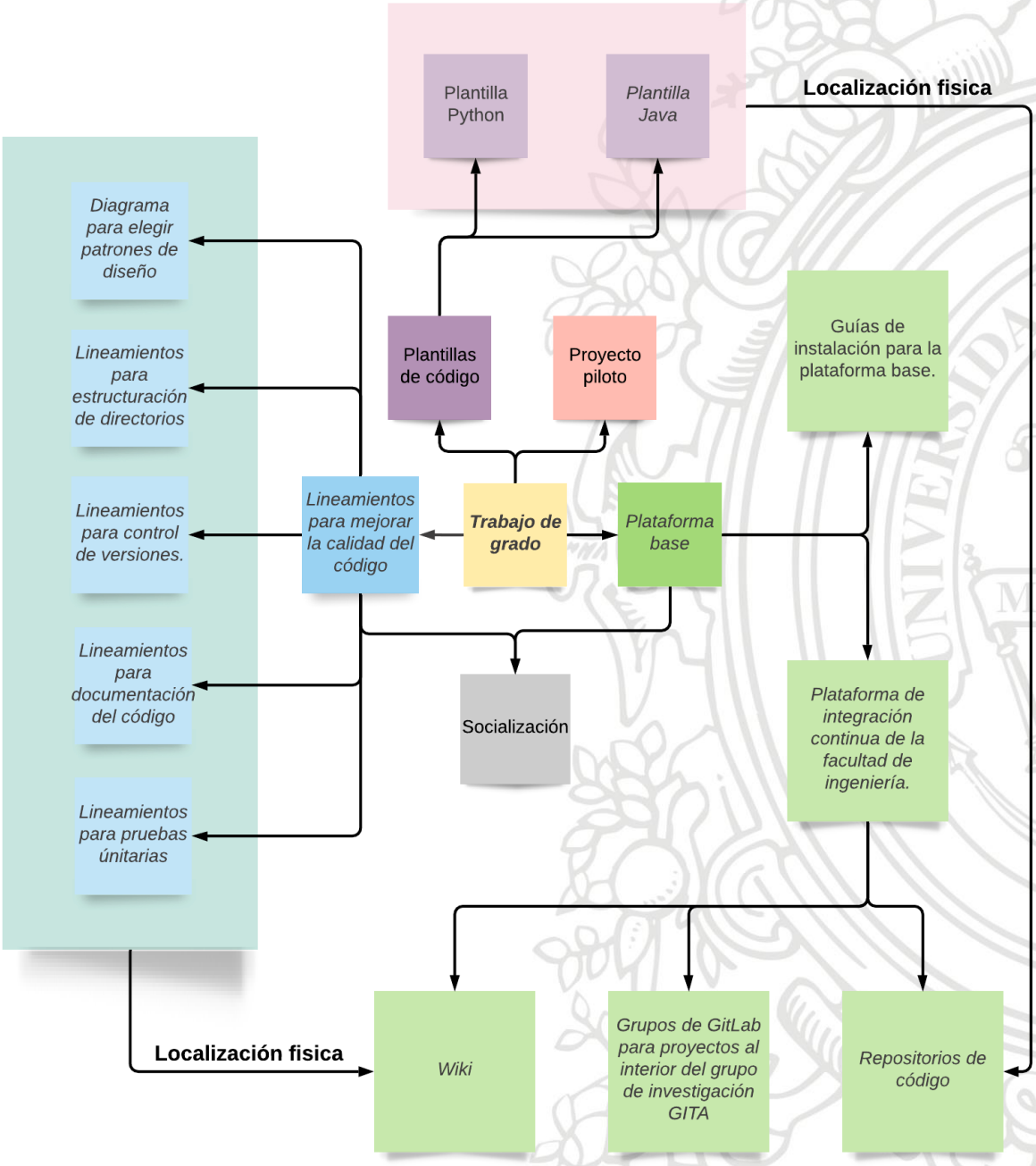

Figura 20. Resumen entregables trabajo de grado

3.1. Plataforma base

La instalación de la plataforma base no se realizó en los servidores del grupo de investigación GITA. Esta decisión se tomó en base a que dicha plataforma demanda un alto consumo de recursos y, a la fecha, los servidores del grupo tienen una limitada capacidad de procesamiento. Sin embargo, ya que la facultad de ingeniería de la Universidad de Antioquia cuenta con una plataforma de integración continúa estructurada, se optó por utilizar la infraestructura de dicha plataforma para el desarrollo de los futuros proyectos del grupo. Aquí es preciso mencionar que esta plataforma solo puede ser utilizada en el ambiente laboral de la universidad.

En este contexto, se realizó todo el proceso de gestión necesario con el Departamento de Recursos de Apoyo e Informática (DRAI), con el propósito de disponer de un espacio para el desarrollo de nuevos proyectos al interior del grupo de investigación GITA.

La coordinación de dicho espacio está a cargo del docente Alexánder Leal, y actualmente 7 desarrolladores del grupo (entre estudiantes de maestría y doctorado) disponen de cuentas para acceder a esta plataforma. La creación de nuevos proyectos y la gestión de los grupos de GitLab se realiza mediante solicitud del docente a cargo dirigida a la ingeniería Clara Monsalve, la cual se desempeña como gestora de proyectos en el DRAI.

3.1.1. Guías de instalación para la plataforma base

El hecho de que los actuales servidores del grupo no dispongan de capacidad de cómputo suficiente para mantener una plataforma base, no significa que en un futuro esta condición no pueda cambiar.

Por lo tanto, para complementar el entregable de la plataforma base se desarrollaron guías de instalación basadas en contenedores. Un contenedor de software es un paquete con unidades funcionales estandarizadas que se encuentra listo para ejecutarse en cualquier sistema operativo. Se caracterizan por su encapsulamiento, mantenibilidad y escalabilidad.

Mediante dichas guías es posible generar una plataforma similar a la usada en la facultad de ingeniería usando Docker como sistema de contenedores.

Las guías contienen: las rutas de los contenedores a ser instalados, los pasos de instalación y anotaciones importantes a tener en cuenta.

3.1.2. Socialización

Uno de los puntos clave para determinar el éxito del trabajo de grado en el futuro del grupo de investigación GITA es la socialización. Para la plataforma base se realizaron integraciones con el grupo para comunicar el funcionamiento de la plataforma base instalada en los servidores de

ingeniería. Además, se socializaron las guías para una posible instalación de la plataforma base en los servidores propios del grupo.

3.2. Documentación con lineamientos para mejorar la calidad del código

Este entregable se encuentra plasmado en el capítulo 2 del presente trabajo de grado. Se resume, en un documento con la definición y estandarización de los lineamientos para la documentación, control de versiones y pruebas unitarias al interior del grupo de investigación GITA.

3.2.1. Socialización

De los lineamientos generados para este trabajo de grado se realizaron 4 reuniones.

1. **Reunión 1 Git/GitLab:** Para esta reunión se trataron los temas de control de versiones, con un espacial énfasis en Git y GitLab.
2. **Reunión 2 Pruebas unitarias:** Para esta reunión se trataron los temas de pruebas unitarias en Python y Java. Entre los ejemplos visualizados se explicaron las plantillas de código que sirven como entregable.
3. **Reunión 3 Estructuración de directorios:** Para esta reunión se trataron los temas de estructuración de directorios y buenas prácticas. Se incluyo un tema extra donde se tocaron los sistemas de administración de dependencias para Java (Maven) y Python (pip).
4. **Reunión 4 Documentación:** Para esta reunión se trataron los temas: calidad en la documentación, comentar vs documentar, cómo se debe pensar al documentar y herramientas para gestionar la documentación.

3.3. Plantillas de código

Este entregable resume los lineamientos propuestos en el capítulo 2 sobre estructuración de directorios. Las plantillas que componente el entregable son: una plantilla en Python y otra en Java que sirven de ejemplo para estructurar proyectos. Además de una breve descripción de su uso. En la sección 2.3.2 se explica con mayor detalle las decisiones de diseño tomadas para seleccionar la estructura de las plantillas.

3.4. Proyecto piloto

Para la prueba piloto se usó el proyecto beermeeting. La razón de peso para este proyecto sobre uno actual del grupo de investigación fue el tiempo de dedicación esperado para este. Dado que mediante el curso de servicios telemáticos era posible unir los lineamientos del trabajo de grado con los entregables del proyecto beermeeting. Además, no se eligió un proyecto del grupo de investigación porque podía generar retrasos en la entrega.

3.4.1. Beermeeting

La guía para el proyecto de **beermeeting** propuesta en el curso de servicios telemáticos tenía la siguiente información [29]:

"Beermeeting (Berra o que?) es un servicio telemático para apoyar la construcción de relaciones entre personas que desean establecer un punto de encuentro para tomar un par de cervezas. Dicho servicio funcionará bajo la arquitectura Cliente/Servidor, integrado dos tipos de estructuras de operación: centralizada (entre el servidor y los clientes) y distribuida (solo entre clientes)."

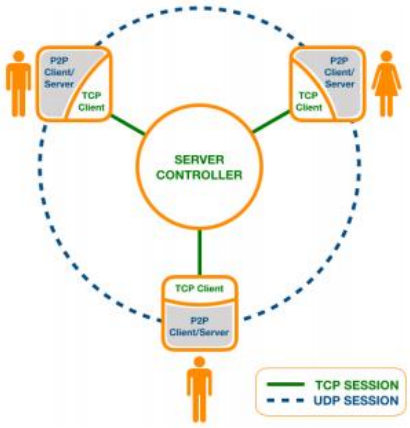

Figura 21. Imagen descriptiva de beermeeting

3.4.1.1. Funcionalidades del servidor

- Almacenar en una base de datos los usuarios que se conectan al sistema, donde cada usuario deberá proporcionar la siguiente información: nombre, edad (18-99), sexo(M/F) y localización (MED, ENV, SAB, EST, BEL, ITA).
- Registrar de manera autónoma cada una de las IP's de los usuarios que se conectan a la plataforma.
- Proveer, cuando el usuario lo requiera, un listado de los demás usuarios conectados indicando su sexo, edad y ubicación.
- Informar a los usuarios en línea la conexión de un nuevo usuario, para que si éstos lo desean vuelvan a pedir la lista de usuarios conectados.

3.4.1.2. Funcionalidades del cliente

- Establecer conexión con el servidor y proveer: nombre, edad (18-99), sexo(M/F) y localización (MED, ENV, SAB, EST, BEL, ITA).
- Solicitar listado de usuarios conectados al servidor
- Estar a la espera de conexión por parte de otros usuarios
- Establecer conexión con otros usuarios
- Garantizar comunicación segura (cifrada) para los mensajes entre usuarios.

3.4.1.3. Entregables:

1. Documentación de la arquitectura del software desarrollado utilizando el modelo de vistas 4+1 de Kruchten.
2. Adicionalmente la documentación debe incluir:
   a. Vocabulario de mensajes usados por el protocolo.
   b. El formato de cada uno de los mensajesdel vocabulario (codificación/ estructura
   c. Las reglas de procedimiento que garantizan la consistencia delintercambio de mensajes.
   d. Carpeta con el código

3.4.1.4. Modificación de entregables

Los proyectos entregados para este curso generalmente contaban con un archivo .zip con el código y una guía en Word que contenía todas las imágenes y pasos necesarios para ejecutar el código. Sin embargo, siguiendo la nueva metodología los entregables cambian. Es de resaltar que el enunciado del proyecto se mantiene pues la metodología no interfiere con los resultados esperados en curso.

Los entregables resultantes son:

1. Documentación de la arquitectura del software desarrollado utilizando el modelo de vistas 4+1 de Kruchten. Esta documentación debe ser incluida dentro la carpeta docs.
2. Documentación dentro del código de métodos, clases y scripts usados.
3. Archivo README.md con nombre del proyecto, descripción corta, descripción detallada (vocabulario de mensajes usados por el protocolo, formato de cada uno de los mensajes del vocabulario (codificación/ estructura) y las reglas de procedimiento que garantizan la consistencia del intercambio de mensajes), librerías usadas en el proyecto, requisitos de uso, pasos de instalación y ejecución y datos de contacto del estudiante.
4. Repositorio con todo el proyecto, el repositorio debe contar con commits que prueben el avance parcial de los estudiantes.
5. Pruebas unitarias que prueben el correcto funcionamiento del proyecto.

Nota: Para el proyecto se debe usar el paradigma de programación orientado a objetos.

3.4.2. Resultados del proyecto

El proyecto raíz de beermeeting reside en el repositorio oficial de la facultad de ingeniería en el enlace:

http://svningenieria.udea.edu.co:8080/apoyo-software-gita/prueba-piloto-gita/network/master

Es de resaltar que para este trabajo de grado se solicitó la creación de un grupo de GitLab llamado Apoyo software GITA, el cual a su vez contiene todos los entregables propuestos para este trabajo de grado incluido el proyecto beermeeting.

En la Figura 22 podemos identificar el resultado final de los directorios del proyecto beermeeting. Inicialmente resalta en la raíz del proyecto los archivos: README.md, requeriments.txt, config.py, dos archivos SQL (para la creación de la base de datos) y el archivo .gitignore. Con excepción de los archivos SQL, todos son parte activa de la metodología y cumplen la importante labor de guiar al usuario para que reconozca que tipo de proyecto está por abrir y como puede usarlo fácilmente. Un proyecto nuevo que use la metodología no debe temer a incluir nuevos archivos que brinden valor agregado en el repositorio principal. Es importante revisar si de verdad amerita ubicar el archivo en la raíz del proyecto. Sin embargo, no es limitante, pues en definitiva la metodología busca facilitar el proceso de desarrollo, no encarecerlo.

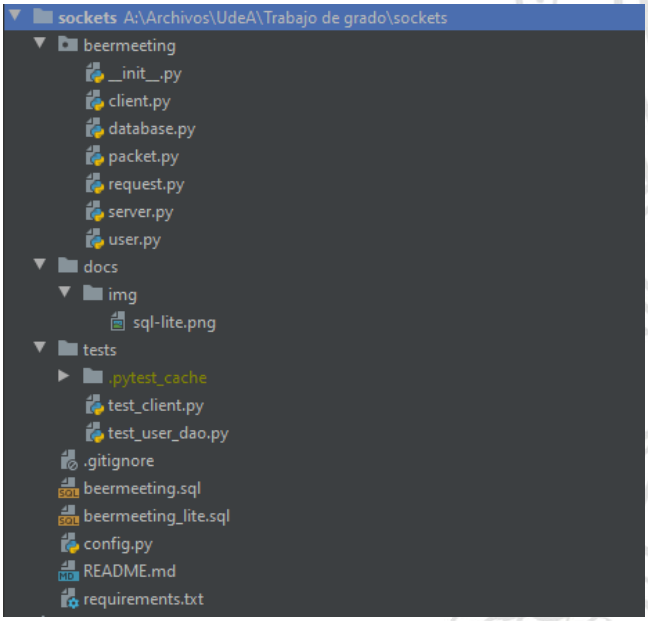

Figura 22. Estructura de directorios proyecto beermeeting

3.4.2.1. Registro de avances y control de versiones

Uno de los puntos clave de la metodología es el uso de Git como controlador de versiones para el proyecto. En el repositorio principal se puede seguir el árbol de commits el cual da una mejor idea del desarrollo del proyecto.

La Figura 23 permite identificar el proceso de desarrollo que se siguió para el proyecto beermeeting. Se puede destacar de la imagen que dicha secuencia de commits no sigue el flujo de trabajo propuesto en el capítulo 2. Esto se debe a que el proyecto solo fue desarrollado por una persona y era pequeño, por lo que el uso de la rama master era suficiente.

La metodología no debe tomarse como flujo rígido que complique el desarrollo más de lo que debe ser, por ende, el flujo de trabajo propuesto en el capítulo 2 puede modificarse, adaptarse e incluso obviarse según la naturaleza del proyecto. Sin embargo, las modificaciones u omisiones deben seguir garantizando las siguientes normas en el proyecto:

- Commits pequeños.
- Datos privados fuera del repositorio.
- Integraciones periódicas.
- Uniones de código limpias.
- Desarrollos independientes fáciles de unir con el código principal.
- Pruebas de valor en el código.
- Datos para replicar el proyecto claros.
- Guías de uso claras y concisas.

Figura 23. Commits en el repositorio de beermeeting

3.4.2.2. Enfoque orientado a objetos

El proyecto debía tener un claro enfoque al paradigma de programación orientado a objetos. Debido a esto todo el código fue implementado mediante objetos que expresaban una entidad única y bien definida en el proyecto. En esta sección se expondrá la parte más importante del código del proyecto beermeeting.

En el siguiente código podemos identificar que existe una amplia documentación la clase BeerServer junto un aislamiento de la lógica para la ejecución del servidor. Esto se debe al principio de única responsabilidad promovido en este trabajo de grado.

```
1.  from socketserver import ThreadingMixIn, TCPServer
2.  from beermeeting.database import Database
3.
4.
5.  class BeerServer(ThreadingMixIn, TCPServer):
6.      """
7.      Servidor principal, este servidor funciona mediante multiples hilos y
    permite la gestión de diferentes clientes
8.      al mismo tiempo. Este servidor también contiene una instancia de base de
    datos configurada para gestionar los
9.      usuarios del sistema.
10.
11.     El servidor usa el driver CX_ORACLE para conectarse al esquema
    beermeeting y gestionar los usuarios que ingresan
12.     al sistema.
13.
14.     En el archivo beermeeting.sql se tiene el esquema necesario para que el
    proyecto funcione de manera local.
15.
16.     """
17.
18.     def __init__(self, server_address, RequestHandlerClass, config,
    bind_and_activate=True):
19.         super().__init__(server_address, RequestHandlerClass,
    bind_and_activate)
20.         Database.get_instance(config.username, config.password, config.dsn)
21.
```

Código 26. Servidor TCP Beermeeting

En el Código 27 podemos visualizar un fragmento de la clase EchoRequestHandler. Mediante esta clase el servidor procesa las peticiones provenientes de los clientes. Es de resaltar que la clase se encuentra completamente documentada junto con sus métodos para permitir al usuario reconocer el objetivo de cada función. También es posible apreciar el uso de programación funcional junto a POO, es decir, múltiples paradigmas en una misma clase.

```
1.  import socketserver
2.  from beermeeting.packet import Packet
3.  from beermeeting.user import User
4.  from beermeeting.user import UserDao
5.  import cx_Oracle
6.  import pickle
7.
8.
9.  class EchoRequestHandler(socketserver.BaseRequestHandler):
10.     """
```

```
11.     Clase para gestionar las peticiones al servidor TCP principal. En esta
    clase se encuentran los metodos para guardar
12.     los usuarios, gestionar los mensajes y verificar las operaciones
    solicitadas por los clientes.
13.
14.     Methods
15.     -------
16.     read_packet()
17.         Esta función permite obtener los paquetes enviados al servidor.
    Escucha las peticiones de los clientes.
18.     handle_error(username: str, error)
19.         Función que gestionar los errores cuando el sistema tiene
    comportamientos no esperados
20.     handle_packets(decode: Packet)
21.         Controla los paquetes entrantes y los tramita según la función que
    realizan en la aplicación
22.     handle_operations(decode: Packet)
23.         Gestiona la operación a realizar en el momento que llega un paquete
    que tramita una operación en el servidor
24.     handle()
25.         Método principal que es sobreescrito de la clase BaseRequestHandler
    para gestionar los peticiones entrantes al
26.         servidor TCP
27.     """
28.
29.     SOCKETS_LIST = []
30.     ACTIVE_USERS = []
31.     TOKEN = "DFGHJKLK"
32.
33.     def read_packet(self) -> Packet:
34.         """
35.         Función para leer los paquetes entrantes
36.         Returns
37.         -------
38.         Packet
39.             Paquete resultante de la petición
40.         """
41.         data = self.request.recv(1024)
42.         return pickle.loads(data)
43.
44.     def handle_error(self, username: str, error):
45.         """
46.         Función para controlar los errores en el servidor
47.
48.         Parameters
49.         ----------
50.         username
51.             Usuario en el cual se presento el error
52.         error
53.             Error en el sistema
54.
55.         """
56.         print("Excepción en el servidor")
57.         print(error)
58.         try:
59.             self.SOCKETS_LIST.remove(self.request)
60.             us = next(filter(lambda usr: usr.username == username,
    self.ACTIVE_USERS), None)
61.             self.ACTIVE_USERS.remove(us)
```

Código 27. Fragmento de código del manejador de peticiones del servidor TCP

El Código 28 es bastante llamativo pues permite al usuario reconocer la manera en cómo el sistema define la base de datos. Además, provee dos implementaciones diferentes de un mismo

padre, Database. De tal manera que las dos implementaciones sobrecargan los métodos del padre y pueden usarse indiscriminadamente en los puntos donde el sistema trata de acceder a la clase Database. Este código es de bastante ayuda para implementaciones dentro del grupo de investigación. Esto, porque define la manera en cómo un sistema debe parametrizarse para aprovechar las ventajas de la programación orientada a objetos.

```
1.  class DatabaseOracle:
2.      """
3.      Clase para gestionar la conexión a la base de datos mediante el driver
    cx_oracle. Esta clase fue modelada con el
4.      patrón de diseño singleton para poder acceder desde un solo punto a la
    base de datos.
5.
6.      Methods
7.      -------
8.      get_instance(username='', password='', dsn='', encoding='UTF-8')
9.          Función para obtener la instancia de la clase Database
10.     cursor()
11.         Retorna el cursor con el cual se realizan las operación en la DB
12.     commit()
13.         Permite realizar un commit de las transacciones realizadas en la db
14.     execute(sql, params=None)
15.         Ejecuta una query con los parámetros definidos
16.     fetchall()
17.         Obtiene los datos gestionados en el cursor
18.     fetchone()
19.         Obtiene un solo dato del cursor
20.     query()
21.         Ejecuta una query y retorna los datos obtenidos en el cursor
22.
23.     """
24.
25.     def __init__(self, username, password, dsn, encoding='UTF-8'):
26.         self.__db = cx_Oracle.connect(username, password, dsn,
    encoding=encoding)
27.         self.__db.autocommit = False
28.         self.__cursor = self.__db.cursor()
29.         Database.__instance = self
30.
31.     def __enter__(self):
32.         return self
33.
34.     def __exit__(self, exc_type, exc_val, exc_tb):
35.         self.__cursor.close()
36.         self.__db.close()
37.
38.     @property
39.     def cursor(self):
40.         return self.__cursor
41.
42.     def commit(self):
43.         self.__db.commit()
44.
45.     def execute(self, sql, params=None):
46.         return self.cursor.execute(sql, params or ())
47.
48.     def fetchall(self):
49.         return self.cursor.fetchall()
50.
51.     def fetchone(self):
52.         return self.cursor.fetchone()
```

```
53.
54.     def query(self, sql, params=None):
55.         self.cursor.execute(sql, params or ())
56.         return self.fetchall()
57.
58.
59. class DatabaseLite:
60.     """
61.     Clase para gestionar la conexión a la base de datos mediante el driver
    cx_oracle. Esta clase fue modelada con el
62.     patrón de diseño singleton para poder acceder desde un solo punto a la
    base de datos.
63.
64.     Methods
65.     -------
66.     get_instance(username='', password='', dsn='', encoding='UTF-8')
67.         Función para obtener la instancia de la clase Database
68.     cursor()
69.         Retorna el cursor con el cual se realizan las operación en la DB
70.     commit()
71.         Permite realizar un commit de las transacciones realizadas en la db
72.     execute(sql, params=None)
73.         Ejecuta una query con los parámetros definidos
74.     fetchall()
75.         Obtiene los datos gestionados en el cursor
76.     fetchone()
77.         Obtiene un solo dato del cursor
78.     query()
79.         Ejecuta una query y retorna los datos obtenidos en el cursor
80.
81.     """
82.
83.     def __init__(self, url):
84.         self.__db = sqlite3.connect(url)
85.         self.__cursor = self.__db.cursor()
86.         DatabaseLite.__instance = self
87.
88.     def __enter__(self):
89.         return self
90.
91.     def __exit__(self, exc_type, exc_val, exc_tb):
92.         self.__cursor.close()
93.         self.__db.close()
94.
95.     @property
96.     def cursor(self):
97.         return self.__cursor
98.
99.     def commit(self):
100.             self.__db.commit()
101.
102.         def execute(self, sql, params=None):
103.             return self.cursor.execute(sql, params or ())
104.
105.         def fetchall(self):
106.             return self.cursor.fetchall()
107.
108.         def fetchone(self):
109.             return self.cursor.fetchone()
110.
111.         def query(self, sql, params=None):
112.             self.cursor.execute(sql, params or ())
113.             return self.fetchall()
```

Código 28. Fragmento de código de las clases para controlar el acceso a la base de datos.

3.4.2.3. Pruebas unitarias

El paso final para cubrir el alcance del proyecto piloto fue la implementación de pruebas para algunos módulos del sistema. Este paso es esencial y tenía como objetivo generar un esquema de ejemplo, para que los usuarios de la metodología tuvieran guías sobre las cuales construir sus primeras pruebas. Es de resaltar que la teoría propuesta en este documento es el eje estructural para la implementación de pruebas. Sin embargo, los ejemplos como éste tienden a ser más ilustrativos cuando se trata de ejecutar una metodología.

Los casos de pruebas en los cuales se centró el proyecto piloto son unitarios, dado que el alcance del proyecto y el tamaño del ejemplo no ameritaban el uso de pruebas de integración o de pruebas de sistema.

En el Código 29 podemos identificar algunas de las pruebas realizadas para el proyecto piloto. En estas se puede visualizar la necesidad de abstraer el comportamiento de la clase Client, para poder probar funcionalidades diferentes sin solapar la prueba.

Es de resaltar que cuando el código es lo suficientemente complejo para no poder generar pruebas unitarias se debe analizar la posibilidad de refactorizar. Puesto que el código no cumple con el principio de única responsabilidad.

```
1.  import unittest
2.  from beermeeting.client import Client
3.  from beermeeting.user import User
4.  from beermeeting.packet import Packet
5.
6.
7.  class TestClient(unittest.TestCase):
8.
9.      def test_register(self):
10.         client = Client()
11.         usr = User("127.0.0.1", 'NIET', 20, 'NICOLAS', 'LN', 'M', 29000)
12.         new_client = Packet(Packet.SERVER_REGISTER, None, None, '', usr)
13.         message = client.process_packet(new_client)
14.         self.assertTrue(message, f"Ingreso con {client.user.username}")
15.
16.     def test_op_get(self):
17.         client = Client()
18.         usr = User("127.0.0.1", 'NIET', 20, 'NICOLAS', 'LN', 'M', 29000)
19.         new_client = Packet(Packet.SERVER_OPERATION, None, None, [usr], usr)
20.         new_client.op = Packet.OP_GET
21.         message = client.process_packet(new_client)
22.         self.assertTrue(len(message), 1)
23.
24.     def test_op_res(self):
25.         client = Client()
26.         usr = User("127.0.0.1", 'NIET', 20, 'NICOLAS', 'LN', 'M', 29000)
27.         packet = Packet(Packet.SERVER_RESPONSE, None, None, "HI BUDDY", usr)
28.         packet.op = Packet.OP_GET
29.         message = client.process_packet(packet)
30.         self.assertTrue(message, "HI BUDDY")
31.
32.     def test_op_error(self):
33.         client = Client()
34.         usr = User("127.0.0.1", 'NIET', 20, 'NICOLAS', 'LN', 'M', 29000)
35.         with self.assertRaises(Exception):
36.             packet = Packet(Packet.SERVER_ERROR, None, None, "HI BUDDY", usr)
37.             client.process_packet(packet)
38.
```

```
39.
40. if __name__ == '__main__':
41.     unittest.main()
42.
```

Código 29. Pruebas unitarias del módulo Client

4. Conclusiones

El grupo de investigación GITA se encuentra en un proceso de mejorar su calidad de desarrollo de software. Mediante este trabajo de grado de espera haber sentado las bases funcionales y teóricas para mejorar la calidad de software con la cual ellos entregan sus códigos. Además de brindar herramientas útiles con las cuales se enfrenten a sus crecientes responsabilidades académicas desde el enfoque productivo del desarrollo de software.

Dentro de los principales avances de calidad de software ingresados al grupo están:

- El uso de repositorios para controlar las versiones y el estado de los documentos.
- La implementación de guías para la construcción de nuevos proyectos con altos estándares de calidad.
- Proyecto guía para apoyar la implementación de la metodología.
- Inclusión del grupo de investigación en los sistemas de integración continua de la facultad de ingeniería.
- Guías para la implementación de una plataforma de integración continua (plataforma base) usando contenedores de GitLab y Jenkins.

Las metodologías que venía usando el grupo de investigación representaban un alto grado de reproceso y dificultad para reusar el código de pares en el grupo. Una gran cantidad del tiempo de construcción de software consistía en la transmisión de conocimiento vía oral, junto con archivos que no contaban con respaldo público o privado. Con la nueva metodología y el proceso de socialización se espera lograr un cambio cultural en la manera de programar, de modo que sea posible dejar constancia escrita del conocimiento. Además de brindar medíos seguros para replicar la información y los códigos desarrollados.

Los procesos de documentación en el grupo eran un tema poco común entre los avances de los diferentes miembros. Cada integrante tenía la potestad de generar un archivo o documento que permitiera a otros desarrolladores replicar sus códigos. No existía documento de requisitos alguno que generara responsabilidad sobre la documentación en el desarrollo. Mediante este trabajo de grado se suplieron parte de dichas falencias, debido a que éste no espera ser un documento de responsabilidad. Sin embargo, si espera ser un documento guía que permita inculcar buenas costumbres y prácticas. De tal manera que la documentación sea generada a medida que el proyecto avanza y exista una trazabilidad completa del estado de cada uno de los nuevos proyectos del grupo de investigación.

Es poco probable que proyectos viejos sean migrados a la nueva metodología. Sin embargo, se generó un marco de trabajo suficientemente estructurado para los nuevos proyectos.

Los procesos de socialización dejaron como aprendizaje que el conocimiento adquirido en las charlas se necesita replicar. Los miembros actuales del grupo de investigación quedaron con los medios y conocimientos necesarios para replicar los estándares de calidad propuestos.

Este trabajo de grado no espera ser la guía definitiva para implementar proyectos de software dentro del grupo. Sin embargo, espera haber dejado una huella lo suficientemente profunda para generar curiosidad y deseo por mejorar la manera en cómo cada miembro del grupo desarrolla sus proyectos de software.

En definitiva, este proyecto de grado cumplió su objetivo al sembrar las bases de los estándares de calidad en los diferentes miembros del grupo de investigación. Es posible que los resultados de este trabajo de grado no puedan visualizarse a corto plazo, dado que el impacto real solo puede identificarse cuando la metodología sea aplicada en proyectos al interior del grupo.

5. Referencias bibliográficas

[1] I. S. 610.12-1990, «729-1983 - IEEE Standard Glossary of Software Engineering Terminology,» 1983.

[2] D. J. Reifer, Software Management, Wiley-IEEE Computer Society Pr; 7 edition, 2014, p. 568.

[3] ISOTool, «Historia y evolución del concepto de Gestión de Calidad,» 30 1 2016. [En línea]. Available: https://www.isotools.org/2016/01/30/historia-y-evolucion-del-concepto-de-gestion-de-calidad/. [Último acceso: 29 03 2020].

[4] ISO, *ISO 9000 family - Quality management*, 2019.

[5] A. V. Feigenbaum, «Total Quality Management,» de *Encyclopedia of Software Engineering*, American Cancer Society, 2002.

[6] J. M. Juran y A. B. Godfrey, Manual de calidad de Juran, McGraw-Hill, Interamericana de España, 2001, pp. 2-3.

[7] M. Mcguire, Programming Language for Compressing Graphics, Springer, Berlin, Heidelberg, 2002.

[8] A. Agarwal, «Object Oriented Programming in C++,» GreekForGreeks, 21 12 2017. [En línea]. Available: https://www.geeksforgeeks.org/object-oriented-programming-in-cpp/#polymorphism. [Último acceso: 04 04 2020].

[9] K. P. N. &. .. G. &. .. S. &. M. R. Reddy, «Comparison of Programming Languages: Review.,» *International Journal of Computer Science & Communication*, pp. 113-122, 2018.

[10] W3 Schools, «Java Introduction,» Refsnes Data, 2 2017. [En línea]. Available: https://www.w3schools.com/java/java_intro.asp. [Último acceso: 29 03 2020].

[11] W3 Schools, «Python Introduction,» Refsnes Data, 2 2017. [En línea]. Available: https://www.w3schools.com/python/python_intro.asp. [Último acceso: 29 03 2020].

[12] D. Young, «Software Development Methodologies,» *White paper*, 8 2013.

[13] W. Cunningham, «Principles behind the Agile Manifesto,» agilemanifesto.org, 02 2001. [En línea]. Available: http://agilemanifesto.org/principles.html. [Último acceso: 29 03 2020].

[14] S. A. I. B. Arachchi y I. Perera, «Continuous Integration and Continuous Delivery Pipeline Automation for Agile Software Project Management,» 2018.

[15] M. Fowler, «Continuos integration,» 01 05 2006. [En línea]. Available: https://martinfowler.com/articles/continuousIntegration.html. [Último acceso: 29 03 2020].

[16] J. Humble y D. Farley, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, 1st ed., Addison-Wesley Professional, 2010.

[17] L. Chen, «Continuous Delivery: Huge Benefits, but Challenges Too,» *IEEE Software*, vol. 32, 3 2015.

[18] A. Shvets, Dive Into DESIGN PATTERNS, 1st ed., https://refactoring.guru/, 2017.

[19] V. Timokhina, *Source code documentation best practices*, 2017.

[20] J. Atwood, Effective Programming: More Than Writing Code, North Charleston, SC, USA: CreateSpace Independent Publishing Platform, 2012.

[21] S. Chacon y B. Straub, Pro Git, 2nd ed., USA: Apress, 2014.

[22] C. R. GitHub Community, «Automation Servers,» Microsoft, 14 11 2016. [En línea]. Available: https://docs.microsoft.com/en-us/cpp/mfc/automation-servers?view=vs-2019. [Último acceso: 29 03 2020].

[23] altexsoft, «Altexsoft,» 19 02 2019. [En línea]. Available: https://www.altexsoft.com/blog/engineering/comparison-of-most-popular-continuous-integration-tools-jenkins-teamcity-bamboo-travis-ci-and-more/. [Último acceso: 09 04 2020].

[24] Jenkins - Community, «Jenkins,» 02 02 2011. [En línea]. Available: https://jenkins.io/. [Último acceso: 09 04 2020].

[25] J. Mertz, *Documenting Python Code: A Complete Guide*, 2019.

[26] Oracle, «Oracle,» 28 02 1997. [En línea]. Available: https://www.oracle.com/technetwork/java/javase/documentation/index-137868.html. [Último acceso: 10 04 2020].

[27] Atlassian, «Bitbucket,» 30 06 2019. [En línea]. Available: https://bitbucket.org/product/es/code-repository. [Último acceso: 09 04 2020].

[28] Atlassian, «Tutorials Git,» Atlassian, 23 10 2017. [En línea]. Available: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow. [Último acceso: 05 04 2020].

[29] D. E. A. L. Piedrahita, «Proyecto beermeeting,» Medellín, 2019.

[30] T. Wasylczenko, «Tips and Tricks for Better Java Docs,» 3 2015. [En línea]. Available: https://www.jrebel.com/blog/tips-and-tricks-for-better-java-documentation. [Último acceso: 29 03 2020].

[31] Solid Studio, «Benefits of CI/CD Pipelines,» 20 08 2017. [En línea]. Available: https://solidstudio.io/blog/ci-cd-pipelines.html. [Último acceso: 29 03 2020].

[32] Rajkumar, «What Is Software Testing – Definition, Types, Methods, Approaches,» Software Testing Material, 13 11 2019. [En línea]. Available: https://www.softwaretestingmaterial.com/software-testing/. [Último acceso: 29 03 2020].

[33] A. Shaw, «Getting Started With Testing in Python,» Real Python, [En línea]. Available: https://realpython.com/python-testing/. [Último acceso: 29 03 2020].

[34] T. Eneh, «Most popular CI/CD pipelines and tools,» Medium, 15 08 2019. [En línea]. Available: https://medium.com/faun/most-popular-ci-cd-pipelines-and-tools-ccfdce429867. [Último acceso: 06 04 2020].

[35] Maven, «Maven,» 13 07 2004. [En línea]. Available: https://maven.apache.org/. [Último acceso: 09 04 2020].

[36] Oracle, «Oracle Technology Network,» 04 2015. [En línea]. Available: https://www.oracle.com/technetwork/es/articles/java/java-con-maven-2516405-esa.html. [Último acceso: 09 04 2020].

[37] Python Packaging Authority, «PyPI,» [En línea]. Available: https://pypi.org/project/setuptools/. [Último acceso: 09 04 2020].

[38] The pip developers, «PyPI,» [En línea]. Available: https://pypi.org/project/pip/. [Último acceso: 09 04 2020].