UNIVERSIDAD ANDINA SIMÓN BOLÍVAR
SEDE CENTRAL

Sucre - Bolivia

MAESTRÍA EN SOFTWARE LIBRE

MARCO DE TRABAJO ÁGIL PARA DESARROLLO DE SOFTWARE LIBRE

Tesis presentada para optar al Grado Académico de Magíster en Software Libre

MAESTRANTE: JOSÉ BORIS BELLIDO SANTA MARÍA

Sucre - Bolivia

2023

UNIVERSIDAD ANDINA SIMÓN BOLÍVAR
SEDE CENTRAL

Sucre - Bolivia

MAESTRÍA EN SOFTWARE LIBRE

MARCO DE TRABAJO ÁGIL PARA DESARROLLO DE SOFTWARE LIBRE

Tesis presentada para optar al Grado Académico de Magíster en Software Libre

MAESTRANTE: JOSÉ BORIS BELLIDO SANTA MARÍA

TUTOR: M.SC. MAURICIO CANSECO TORRES

Sucre - Bolivia

2023

DEDICATORIA

A José y Yolanda, mis padres.

A Modesta, mi segunda mamá.

A Alvaro y Dennis Jesús, mis hermanos.

AGRADECIMIENTOS

A la Universidad Andina Simón Bolívar por propiciar este programa de postgrado para el mejoramiento y crecimiento personal y profesional.

A mi tutor el M.Sc. Mauricio Canseco Torres por todo el apoyo y ayuda recibidos.

A los responsables de equipos y a los desarrolladores que han participado en el diagnóstico.

A los expertos que han validado la propuesta.

A mi colega el M.Sc. Windsor Alvarez por el empujón final.

RESUMEN

Bolivia migrará todo el software del estado a Software Libre hasta 2024, por lo tanto es necesario desarrollar muchos software en poco tiempo, en Sucre la mayoría de los equipos de desarrollo son muy pequeños y serán ellos los encargados de desarrollar gran parte del software nuevo. Las metodologías y marcos de trabajo para el desarrollo de software no están pensado para equipos pequeños por lo tanto es difícil adaptarlos para que sean un aporte importante al trabajo de desarrollo de software. El problema es ¿cómo mejorar el proceso de desarrollo de Software Libre de equipos muy pequeños? para dar respuesta el objetivo general es Diseñar un marco de trabajo ágil para mejorar el proceso de desarrollo Software Libre de equipos muy pequeños.

El problema es ¿cómo mejorar el proceso de desarrollo de Software Libre de equipos muy pequeños? para dar respuesta el objetivo general es Diseñar un marco de trabajo ágil para mejorar el proceso de desarrollo Software Libre de equipos muy pequeños; para lograr dicho objetivo se plantean los siguientes objetivos específicos: Sistematizar los principales referentes procesos de desarrollo ágil de software. Valorar de forma crítica las características actuales del proceso de desarrollo de software de los equipos muy pequeños. Identificar los componentes de un marco de trabajo ágil que permiten mejorar el proceso de desarrollo de Software Libre de los equipos muy pequeños. Determinar el grado de viabilidad y validez del marco de trabajo ágil para mejorar el proceso de desarrollo de Software Libre de equipos muy pequeños.

Se han sistematizado los principales referentes teóricos de metodologías ágiles que no definen roles. Para caracterizar el proceso de desarrollo actual se han realizado encuestas desarrolladores y entrevista a responsables de equipo que han participado en el desarrollo de Software Libre. Se han identificado los componentes de un marco de trabajo ágil que puedan ayudar a mejorar el proceso de desarrollo de Software Libre. Los expertos han validado todos los aspectos de la propuesta de manera favorable.

Se han cumplido con todos los objetivos planteados, se recomienda estudiar a las comunidades para que crezca de manera saludable y explorar estrategias para mejorar el procesos

de aseguramiento de la calidad del software.

La comunidad es el aspecto más importante en un proyecto de Software Libre, es importante conocer diferentes aspectos de la misma. Es necesario saber las herramientas de comunicación asíncronas y síncronas y el protocolo de comunicación. Por otro lado, se requiere comprender todos los elementos técnicos y herramientas relacionadas al desarrollo de software. Además, es imprescindible conocer las versiones disponibles, el mantenimiento que reciben y el tiempo de vigencia. Finalmente, se demanda conocer y entender la licencia y todos los aspectos legales relaciones a la propiedad intelectual.

Las actividades se enmarcan en tres momentos:

- Implementación.
- Especificación.
- Exploración.

Los artefactos de software que son necesarios para cumplir con las actividades son:

- Historias de usuario.
- Código fuente.
- Tablero Kanban.

Palabras clave desarrollo ágil, software libre, equipos muy pequeños.

ÍNDICE GENERAL

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

INTRODUCCIÓN

El desarrollo de software es una de las actividades más importantes de la sociedad actual, el Software Libre se ha convertido en un componente esencial en los procesos de muchas instituciones públicas y privadas. Es necesario desarrollar software calidad, que no solo resuelva el problema de los usuarios sino que además permita ser modificado, mejorado y adecuado en el futuro a las nuevas necesidades y exigencias; para conseguirlo es necesario usar la Ingeniería de Software para garantizar la calidad de todos los artefactos y entregables producidos.

Dentro de la Ingeniería de software existen muchas maneras de organizar a los equipos, definiendo roles, actividades y responsabilidades; pero en su gran mayoría están pensadas para equipos grandes. Los equipos pequeños que tratan de adaptar dichas metodologías y marcos de trabajos generalmente fracasan y terminan haciendo el software de manera artesanal sin ningún proceso de Ingeniería de Software y mucho menos para el desarrollo de Software Libre. Se pretende aportar a la Ingeniería de software con una alternativa para equipos muy pequeños.

El problema es ¿cómo mejorar el proceso de desarrollo de Software Libre de equipos muy pequeños? para dar respuesta el objetivo general es Diseñar un marco de trabajo ágil para mejorar el proceso de desarrollo Software Libre de equipos muy pequeños.

El contenido de la presente tesis se organiza en los siguientes capítulos:

I. Aspectos generales.

II. Marco teórico.

III. Metodología de investigación.

IV. Resultados, conclusiones y recomendaciones de la investigación.

V. Propuesta de mejoramiento.

CAPÍTULO I

ASPECTOS GENERALES

1.1 ANTECEDENTES

Sin computadoras el mundo actual es difícil de entender, las mismas han afectado todas las actividades de la vida moderna; las personas han incluido a las computadoras en todas sus actividades, como ser: el trabajo, el ocio, la comunicación, la educación y la información. Las nuevas técnicas y métodos de producción de hardware han permito que los costos de las computadoras sean cada vez más bajos, permitiendo a las personas adquirirlos de manera más fácil; por otro lado, las computadoras se han hecho más pequeñas permitiendo a los fabricantes incluirlas en diferentes electrodomésticos, dotando a los mismo de la capacidad de conectarse a una red e interactuar entre ellos.

Sin duda, que las personas tengan acceso a hardware es muy importante para la democratización de las computadoras, pero no es suficiente. El software permite aprovechar y usar de manera fácil el hardware, sin la necesidad de entender las complejidades del mismo. El software hace que las personas con poca capacitación puedan usar las computadoras y otros dispositivos de manera sencilla y agradable.

La investigación y el desarrollo del hardware son actividades que realizan pocas instituciones porque es necesario contar con equipos e infraestructuras costosos. Por otro lado, para la investigación y desarrollo del software solo es necesario una computadora y acceso a Internet, con lo cual los costos se reducen significativamente. El software libre ha permitido que el conocimiento asociado al software pueda estar disponible para cualquier persona de manera fácil, permitiendo a todo aquel que lo desee pueda usar, estudiar, modificar y compartir el software en forma sencilla y legal.

La Ley General de Telecomunicaciones, Tecnologías de Información y Comunicación Bolivia (2011) en su artículo 77 dice: “Los Órganos Ejecutivo, Legislativo, Judicial y Electoral en todos sus niveles, promoverán y priorizarán la utilización del software libre

y estándares abiertos, en el marco de la soberanía y seguridad nacional", es decir que el estado tomará como primera opción y de manera preferente las soluciones de software libre y solo en casos donde el software no pueda cumplir con las necesidades requeridas se optará por software propietario.

1.2 PLANTEAMIENTO DEL PROBLEMA

El Plan de Implementación de Software Libre y Estándares Abiertos señala la migración de todo el software usado por las instituciones públicas, es decir que el software que actualmente esta funcionando dentro de los procesos institucionales debe ser migrado a Software Libre y usando estándares abiertos en siete años, esto implica un gran esfuerzo por parte de todas las instituciones es vista que mucho de ese software se ha hecho sin seguir procedimientos ingenieriles y en algunos casos no se encuentra adecuadamente documentado.

La migración a software libre implica el desarrollo desde cero, pues el software actual ha sido desarrollado a medida cumpliendo las peculiaridades y la legislación boliviana, por lo que es difícil encontrar algún proyecto que se pueda utilizar directamente. Si bien en algunas instituciones cuentan con equipos de desarrollo la gran mayoría disponen con personal de soporte a usuario final y de infraestructura, siendo el desarrollado de software encargado a terceros ya sea a empresas o consultores.

Los gobiernos departamentales y municipales deben prever las acciones necesarias para cumplir con la migración ya sea con la contratación de empleados, consultores o empresas externas que pueda proveer de todo el software necesario para que la institución siga desarrollando sus actividades con normalidad.

Las empresas grandes de desarrollo de software del país suelen trabajar para empresas del extranjero y si bien el estado siempre es buen cliente, la burocracia y lentitud de los procesos hacen que las empresas eviten trabajar con el estado en proyectos de migración, dichas empresas se concentran en La Paz, Cochabamba y Santa Cruz. En Sucre la mayoría de las empresas de desarrollo de software tiene poco personal. Las empresas que tendrán en sus manos el desarrollo de software para muchas instituciones públicas del municipio

y departamento son empresas micro y pequeñas empresas que cuentan con pocos empleados.

Para desarrollar software de manera eficiente es necesario que el equipo se organice de alguna manera, las metodologías y marcos de trabajo para el desarrollo de software ayudan con esta labor, dando lineamientos de los roles, actividades y responsabilidad que debe asumir cada miembro del equipo para lograr producir un producto de software que satisfaga las necesidades de los usuarios y cumpla con las expectativas de los clientes. En equipos muy pequeños el difícil seguir las directrices de los marcos de trabajo, en vista que una persona debe desempeñar muchos roles; en algunos casos dichos roles, según la teoría, no los puede desempeñar la misma persona.

Al no poderse aplicar adecuadamente el marco de trabajo, en los equipos muy pequeños lo que tiene mayor importancia es el producto y no el proceso. A los equipos les interesa entregar el software más pronto posible, esperando no tener errores y que los usuarios y clientes estén satisfechos con las funcionalidades entregadas.

1.2.1. Formulación del problema científico

¿Cómo mejorar el proceso de desarrollo de Software Libre de equipos muy pequeños?

1.3 LA HIPÓTESIS

Si se aplica un marco de trabajo ágil para equipos muy pequeños entonces mejorará el proceso de desarrollo de Software Libre.

Definición conceptual de variables se presenta en la tabla I.1.

Tabla I.1: Definición conceptual de variables

Variable	Tipo	Definición	Indicadores
Marco de trabajo ágil para equipos muy pequeños	Independiente	Es el conjunto de reglas y prácticas que deben seguir los equipos para desarrollar software, de tan manera que estén dispuestos al cambio.	▪ Actividades. ▪ Artefactos. ▪ Herramientas.
Proceso de desarrollo de Software Libre	Dependiente	Es el conjunto de actividades que realiza un equipo para convertir los requerimientos en software libre funcional.	▪ Organización del equipo. ▪ Artefactos utilizados. ▪ Herramientas usadas.

Fuente: Elaboración propia.

1.4 OBJETIVOS GENERAL Y ESPECÍFICOS

1.4.1. Objetivo general

Diseñar un marco de trabajo ágil para mejorar el proceso de desarrollo Software Libre de equipos muy pequeños.

1.4.2. Objetivos específicos

- Sistematizar los principales referentes procesos de desarrollo ágil de software.
- Valorar de forma crítica las características actuales del proceso de desarrollo de software de los equipos muy pequeños.
- Identificar los componentes de un marco de trabajo ágil que permiten mejorar el proceso de desarrollo de Software Libre de los equipos muy pequeños.
- Determinar el grado de viabilidad y validez del marco de trabajo ágil para mejorar el proceso de desarrollo de Software Libre de equipos muy pequeños.

1.5 OPERACIONALIZACIÓN DE LOS OBJETIVOS DE ESTUDIO

La operacionalización de los objetivos se puede ver en la tabla I.2.

Tabla I.2: Operacionalización de los Objetivos de Estudio

Objetivos	Variables y Dimensiones	Instrumento de recolección de datos
Sistematizar los principales referentes procesos de desarrollo ágil de software.	Roles. ▪ Cantidad. ▪ Responsabilidad. Actividades. ▪ En equipo. ▪ En solitario. Artefactos. ▪ Propósito. Herramientas. ▪ Propósito.	Revisión bibliográfica.

Continua en la página siguiente.

Objetivos	Variables y Dimensiones	Instrumento de recolección de datos
Valorar de forma crítica las características actuales del proceso de desarrollo de software de los equipos muy pequeños.	Organización del equipo. ▪ Metodología usada. ▪ Cantidad de miembros. ▪ Roles. ▪ Tiempo destinado para la planificación. ▪ Tiempo destinado para la programación. ▪ Tiempo destinado para las pruebas. ▪ Tiempo destinado con el cliente y los usuarios. ▪ Herramientas usadas.	Encuesta.
Determinar el grado de viabilidad y validez del marco de trabajo ágil para mejorar el proceso de desarrollo de Software Libre de equipos muy pequeños.	Viabilidad ▪ Técnica. ▪ Económica. Validez ▪ Pertinencia de las actividades. ▪ Pertinencia de los artefactos.	Delphi

Fuente: Elaboración propia.

1.6 ALCANCES DE LA INVESTIGACIÓN

La investigación se centrará en los desarrolladores de la ciudad de Sucre.

Solo se tomará en cuenta equipos de desarrollo de uno a tres miembros que desarrollen software.

CAPÍTULO II

MARCO TEÓRICO

2.1 DESARROLLO DE SOFTWARE

La elaboración de software de computadora es un proceso reiterativo de aprendizaje social, y el resultado es la reunión de conocimiento recabado, depurado y organizado a medida que se realiza el proceso (Pressman, 2010). El desarrollo de software es un proceso social donde el cliente, los usuarios y el equipo de desarrollo aprenden mutuamente a comunicarse y resolver problemas para obtener un producto de software que permita a todos los involucrados sentirse satisfechos del resultado.

Los equipos de desarrollo de software están integrados por personas muy talentosas y con personalidades diferentes. Machuca-Villegas et al. (2021) sostienen que, "el producto de software es el resultado de la ejecución de actividades de un equipo de trabajo. Este equipo debe tener y desarrollar habilidades para la resolución de problemas, aspectos cognitivos e interacción social". Los integrantes del equipo deben trabajar juntos para alcanzar la meta de entregar un producto de software de calidad. Resolver los problemas internos del equipo y tener una adecuada interacción entre los miembros es muy importante para el éxito del proyecto.

Pressman (2010) sostiene que "la ingeniería de software es llevada a cabo por personas creativas y preparadas que deben adaptar un proceso maduro de software a fin de que resulte apropiado para los productos que construyen y para las demandas de su mercado". La creatividad de las personas permite adaptar los métodos de la ingeniería de software a las necesidades del cliente y de los usuarios para construir productos de los cuales no se tiene referencia y es necesario que cumplan con las exigencias de un mercado y una sociedad cada vez más informatizada y exigente.

Las personas son el componente más importante de cualquier proyecto de desarrollo de software, un equipo fuerte es capaz de vencer cualquier dificultad y alcanzar sus objetivos.

2.2 DESARROLLO ÁGIL

2.2.1. Manifiesto ágil

El desarrollo de software ha sufrido muchos cambios a lo largo del tiempo, las primeras aproximaciones a una metodología de desarrollo se han basado en otros ingenierías donde cada fase esta claramente definida y documentada, con roles especializados en cada una de estas fases. A diferencia de otras ingenierías el desarrollo de software tiene mucha incertidumbre, por lo tanto es difícil terminar una fase de manera completa para continuar con la siguiente de forma exitosa.

En vista que las metodologías adaptadas de otras ingenierías no daban buenos resultados y la gran preocupación sobre desarrollar software con calidad; respetando el calendario, el presupuesto y alcance del proyecto. Uribe and Ayala (2007) cuentan que:

> En febrero de 2001 en Utah-EEUU, se reunieron 17 empresarios de la industria del software y como resultado del debate respecto a las metodologías, principios y valores que deben regir el desarrollo de software de buena calidad, en tiempos cortos y flexible a los cambios, se aceptó el término ágil para hacer referencia a nuevos enfoques metodológicos en el desarrollo de software.

El desarrollo de software ágil acepta la incertidumbre como un aspecto más dentro del proceso de desarrollo, por lo tanto los cambios son bienvenidos. Cada aspecto y funcionalidad del software que debe cambiarse, cuando el proyecto ya ha iniciado, debe ayudar a cumplir con las exigencias y expectativas de todos los involucrado.

Uribe and Ayala (2007) afirman que el manifiesto ágil es un “documento que resume en cuatro valores y 12 principios las mejores prácticas para el desarrollo de software, basados en la experiencia de 17 industriales del software, en procura de desarrollos más rápidos conservando su calidad”. La experiencia en la industria del desarrollo de software da como resultado el manifiesto ágil que pretende dar directrices a otros desarrolladores para afrontar proyectos de desarrollo de software, dando como resultado software con calidad en el tiempo y presupuesto acordados.

El manifiesto ágil, redactado por Beck et al. (2001a), dice:

> Estamos descubriendo formas mejores de desarrollar software tanto por nuestra propia experiencia como ayudando a terceros. A través de este trabajo hemos aprendido a valorar:
>
> - **Individuos e interacciones** sobre procesos y herramientas.
> - **Software funcionando** sobre documentación extensiva.
> - **Colaboración con el cliente** sobre negociación contractual.
> - **Respuesta ante el cambio** sobre seguir un plan.
>
> Esto es, aunque valoramos los elementos de la derecha, valoramos más los de la izquierda.

Los aspectos más importantes son las personas y como interactúan entre sí, al final el desarrollo de software es una actividad humana, donde unas personas, los usuarios y clientes, cuentan sus necesidades, ambiciones, y miedos; por otro lado otras personas, el equipo de desarrollo, intenta producir un software que ayude las personas a cubrir su necesidades que les ayuda a alcanzar sus ambiciones y minimice sus miedos. La mejor manera de validar que el equipo de desarrollo va por buen camino es el software funcionando, los usuarios pueden usar el software probar las funcionalidades implementadas y dar retroalimentación para que mantener lo bueno y mejorar lo malo, para llegar lo más pronto posible al producto esperado por todos. El cambio sin duda está presente en este frecuente intercambio de software funcional y retroalimentación, que permite desarrollar un software donde todos los involucrados, clientes, usuarios y equipo, queden satisfechos.

El manifiesto ágil sigue doce principios (Beck et al., 2001b):

1. Nuestra mayor prioridad es satisfacer al cliente mediante la entrega temprana y continua de software con valor.
2. Aceptamos que los requisitos cambien, incluso en etapas tardías del desarrollo. Los procesos Ágiles aprovechan el cambio para proporcionar ventaja competitiva al cliente.

3. Entregamos software funcional frecuentemente, entre dos semanas y dos meses, con preferencia al periodo de tiempo más corto posible.
4. Los responsables de negocio y los desarrolladores trabajamos juntos de forma cotidiana durante todo el proyecto.
5. Los proyectos se desarrollan en torno a individuos motivados. Hay que darles el entorno y el apoyo que necesitan, y confiarles la ejecución del trabajo.
6. El método más eficiente y efectivo de comunicar información al equipo de desarrollo y entre sus miembros es la conversación cara a cara.
7. El software funcionando es la medida principal de progreso.
8. Los procesos Ágiles promueven el desarrollo sostenible. Los promotores, desarrolladores y usuarios debemos ser capaces de mantener un ritmo constante de forma indefinida.
9. La atención continua a la excelencia técnica y al buen diseño mejora la Agilidad.
10. La simplicidad, o el arte de maximizar la cantidad de trabajo no realizado, es esencial.
11. Las mejores arquitecturas, requisitos y diseños emergen de equipos autoorganizados.
12. A intervalos regulares el equipo reflexiona sobre cómo ser más efectivo para a continuación ajustar y perfeccionar su comportamiento en consecuencia.

Los principios dan lineamientos de como deben ser los equipos y sus miembros para conseguir un proceso ágil. Por otro lado, el software funcionando se constituye en el artefacto más importante del proceso. Además, la interacción con los usuarios y el cliente es muy importante para el éxito del proyecto.

2.2.2. Metodologías

Desarrollo guiado por pruebas de software

El desarrollo guiado por pruebas de software, o *Test-driven development (TDD)* es una metodología de desarrollo de software en la que se escriben pruebas para guiar la escritura del código de producción (Iglesias, 2021). En lugar de dejar las pruebas para el final, tarea que en muchas ocasiones no se realiza de manera correcta, la prueba es el primer paso para implementar una funcionalidad.

> Las pruebas especifican de manera formal, ejecutable y mediante ejemplos, los comportamientos que debe realizar el software que estamos programando, definiendo pequeños objetivos que, al ser superados, nos permiten construir el software de forma progresiva, segura y estructurada.
>
> Aunque hablemos de pruebas, no estamos hablando de *Quality Assurance* (QA). (Iglesias, 2021).

Escribir las pruebas antes del código de la aplicación permiten capturar de mejorar manera los requerimientos la identificar los comportamientos que debe realizar la funcionalidad permitida. Las pruebas deben ayudar a marcar objetivos para generar incrementos constantes en el desarrollo. Al ser un proceso incremental, las pruebas se deben revisar y ajustar las veces que sean necesarias.

> Tanto TDD como QA se basan en la utilización de los pruebas como herramientas, pero este uso se diferencia en varios aspectos. Específicamente, en TDD:
>
> - La prueba se escribe antes de que el software que ejecuta siquiera exista.
> - Las pruebas son muy pequeños y su objetivo es forzar la escritura del código de producción mínimo necesario para que la prueba pase, que tiene el efecto de implementar el comportamiento definido por la prueba.
> - Las pruebas guían el desarrollo del código y el proceso contribuye al

diseño del sistema.

(Iglesias, 2021).

Tanto las pruebas como el código son escritos por el programador que implementa dicha funcionalidad. Las pruebas ayudan a que el desarrollador tenga claro lo que debe implementar y se concentre en ello y no perder tiempo y recursos en casos que posiblemente no son requeridos por el cliente y los usuarios.

> En TDD las pruebas se escriben en una forma que podríamos considerar como de diálogo con el código de producción. Básicamente se trata de:
>
> - Escribir una prueba que falla.
> - Escribir código que haga que la prueba pase.
> - Mejorar la estructura del código (y de la prueba).
>
> (Iglesias, 2021).

El dialogo entre las pruebas y el código permite el desarrollo de software con mejor calidad, en vista que todo el software se encuentra probado incluso antes de ser escrito. Por otro lado, las pruebas permiten mejorar el software de manera segura disminuyendo de la probabilidad de introducir errores al momento de modificar el código.

> Una vez que tenemos claro la pieza de software en la que vamos a trabajar y la funcionalidad que queremos implementar, lo primero es definir una primer prueba muy pequeña que fallará sin remedio porque ni siquiera existe un archivo que contenga el código de producción necesario para que se pueda ejecutar (Iglesias, 2021).

En muchas ocasiones el desarrollador asumen que ha entendido correctamente la funcionalidad que debe implementar, pero no en realidad no ha entendido adecuadamente el requerimiento y no sabe como iniciar el trabajo; escribir las prueba permite al desarrollar verificar que realmente ha entendido lo suficiente como comenzar su trabajo, la prueba debe ser lo más simple posible y probar un aspecto que el desarrollador tiene claro cómo cumplirlo.

“En TDD es fundamental ver que las pruebas fallan, no basta con suponerlo. Nuestro trabajo es hacer que la prueba falle por la razón correcta y luego hacerla pasar escribiendo código de producción” (Iglesias, 2021). Las pruebas deben fallar por la razón correcta, es necesario ejecutar la prueba y analizar el resultado de la misma, la falla resultante debe ser esperada por el desarrollador. Las fallas por acceso a archivos, por sintaxis y uso apropiado del lenguaje de programación se deben mitigar como parte de la escritura de la prueba. La única falla que se debe esperar es la referida a la funcionada que se están desarrollando.

Con la prueba concluida se procede a implementar la funcionalidad, “se escribe el código de producción necesario para que la prueba pase, pero nada más” (Iglesias, 2021). Es necesario que el desarrollador se concentre y escriba únicamente el código requerido por la prueba, para ello debe ejecutar las veces que haga falta hasta que no falle, es decir hasta que el código pase la prueba.

“Cuando se ha logrado hacer pasar cada prueba debemos examinar el trabajo realizado hasta el momento y comprobar si es posible refactorizar tanto el código de producción como el de la prueba” (Iglesias, 2021). Con la prueba y el código se ha conseguido cumplir con la funcionalidad, pero que funcione no quiere decir que sea la mejor manera de hacerlo, con los primeros pasos el desarrollador se ha concentrado en encontrar un camino que de solución al problema. Sabiendo que la funcionalidad cumple con el requisito es tiempo de controlar la calidad de la prueba y del código, para lo cual es necesario verificar si se están aplicando todas las reglas y estándares de codificación; también es recomendable refactorizar y optimizar la prueba y el código; y finalmente identificar algunas más prácticas y subsanarlas, todo con el fin de producir una prueba y un código con alta calidad.

Al terminar con el desarrollo de una parte de la funcionalidad, es decir que pase satisfactoriamente la prueba se debe seguir con otro aspecto de la funcionalidad repitiendo el mismo proceso, “una vez que el código de producción hace pasar la prueba y está lo mejor organizado posible en esa fase, es el turno de escoger otro aspecto de la funcionalidad y crear una nueva prueba que falle para describirlo” (Iglesias, 2021). Con cada incremento en la funcionalidad que se agrega se tiene la oportunidad de mejorar el código con la

optimización y refactorización del mismo.

“Existe una manera más formal de asegurarnos de que una funcionalidad está completa. Básicamente consiste en no ser capaz de crear una nueva prueba que falle” (Iglesias, 2021). La funcionalidad ha sido terminada cuando todos los aspectos del requisito han sido probados e implementados satisfactoriamente. Con la funcionalidad terminada se puede seguir con la siguiente hasta completar con el desarrollo del software.

Desarrollo guiado por pruebas de aceptación

El desarrollo guiado por pruebas de aceptación, o *Acceptance Test-Driven Development (ATDD)* es una práctica en la que todo el equipo, destacando, e incluyendo, a los usuarios, desarrolladores y probadores, analiza conjuntamente los criterios de aceptación, antes de que comience el desarrollo (Garzas, 2015). Las pruebas no solo se realizan desde la perspectiva del desarrollador que va implementar la funcionalidad, se hacen desde el punto de vista de todos los involucrados. “El algoritmo es el mismo de tres pasos pero son de mayor zancada que en el TDD practicado exclusivamente por desarrolladores” (Blé, 2010).

> ATDD involucra a miembros del equipo con diferentes perspectivas (cliente, desarrolladores, probadores) que colaboran para escribir pruebas de aceptación antes de implementar la funcionalidad correspondiente. Las discusiones colaborativas que ocurren para generar la prueba de aceptación a menudo se denominan los tres amigos, que representan las tres perspectivas del cliente (¿qué problema estamos tratando de resolver?), desarrolladores (¿cómo podemos resolver este problema?) y probadores (¿qué pasa con esto, qué podría pasar?). Estas pruebas de aceptación representan el punto de vista del usuario y actúan como una forma de requisitos para describir cómo funcionará el sistema, además de servir como una forma de verificar que el sistema funciona según lo previsto. En algunos casos, el equipo automatiza las pruebas de aceptación (Agile Alliance, 2022).

Definir pruebas buenas es vital para guiar adecuadamente el desarrollo del software, ayu-

dan a entender las necesidades del cliente y por otro lado apoyan en la verificación de que el software cumple con dichas necesidades. Las pruebas de aceptación se definen desde la visión del cliente, es decir la dificultad que tienen en sus actividades que se espera que el software ayude a mitigar dicha dificultad; la visión del desarrollador, es decir entender plenamente el requerimiento, por otro lado contemplar la viabilidad de la solución en función de la tecnología y recursos disponibles, finalmente la visión de probador, es decir cuestionar escenario que no son del dominio del problema pero pueden incidir en el producto final, como ser la accesibilidad y el rendimiento.

Tener claro lo que quiere y necesitan los usuarios y el cliente es muy importante al momento de desarrollar software, una buena manera de entender los requerimientos es a través de ejemplos que proporcionan los usuarios y el cliente, los cuales sirven para entender el dominio del problema y luego para validar que el producto resultante satisface las expectativas de los usuarios y cliente. “Las pruebas de aceptación o de cliente son el criterio escrito de que el software cumple los requisitos de negocio que el cliente demanda. Son ejemplos escritos por los dueños de producto” (Blé, 2010).

En ATDD para modelar los requerimientos se puede usar historias de usuario, “cada historia de usuario contiene una lista de ejemplos que cuentan lo que el cliente quiere, con total claridad y ninguna ambigüedad” (Blé, 2010). Cada historia de usuario puede tener varias pruebas de aceptación asociadas a la misma que permitan verificar si el software realmente cumple con todo el comportamiento esperado. Según Blé (2010), cada historia provoca una serie de preguntas acerca de los múltiples contextos en que se puede dar; son las que naturalmente hacen los desarrolladores al cliente; las respuestas son afirmaciones, ejemplos, las cuales se transforman en pruebas de aceptación.

> Para cada prueba de aceptación de una historia de usuario, habrá un conjunto de pruebas unitarias y de integración de grano más fino que se encargará, primero, de ayudar a diseñar el software y, segundo, de afirmar que funciona como sus creadores querían que funcionase (Blé, 2010).

Cada prueba de aceptación puede desencadenar la definición de otras pruebas que son necesarias tanto para definir la estructura interna del software, como la evolución del software para cumplir con los deseos de los usuarios y cliente.

Según Blé (2010) es necesario “desarrollar nuestra habilidad de preguntar al cliente qué quiere y no cómo lo quiere; evitamos a toda costa ejemplos que se meten en el cómo hacer, más allá del qué hacer”. En muchas ocasiones el dialogo con los usuarios y el cliente se va dirige al cómo desea el producto y no a lo que realmente necesitan. Las pruebas de aceptación están pensadas para verificar el cumplimiento de la funcionalidad que se requiere y no a como se la va realizar. Tener claro lo que se solicita permite a los desarrolladores encontrar la mejorar manera de realizarlo, aprovechando las fortalezas de las herramientas y tecnología usadas en el proyecto.

Blé (2010) afirma, una ventaja de ATDD es que vamos a poder comprobar muy rápido si el programa está cumpliendo los objetivos o no; conocemos en qué punto estamos y cómo vamos progresando. Las pruebas de aceptación permiten de manera objetiva evaluar el avance del proyecto y si la finalidad del mismo se está cumpliendo. Por otro lado, aseguran la calidad del software, entregando un producto probado minuciosamente.

> Fuera del contexto ágil, ATDD tiene pocas probabilidades de éxito ya que si los analistas no trabajan estrechamente con los desarrolladores y probadores, no se podrá originar un flujo de comunicación suficientemente rico como para que las preguntas y respuestas aporten valor al negocio.
>
> Si en lugar de ejemplos, se siguen escribiendo descripciones, estaremos aumentando la cantidad de trabajo considerablemente con lo cual el aumento de coste puede no retornar la inversión. Si los dueños de producto (cliente y analistas) no tienen tiempo para definir las pruebas de aceptación, no tiene sentido encargárselos a los desarrolladores, sería malgastar el dinero. Tener tiempo es un asunto muy relativo y muy delicado (Blé, 2010).

El desarrollo ágil permite que ATDD se pueda desarrollar plenamente en vista que los usuarios y clientes deben estar igual de involucrados en el proyecto que el equipo de desarrollado; sin las colaboración estrega de los usuarios, programadores y probadores no se pueden definir correctamente las pruebas de aceptación que guían a todo el proceso de desarrollo de software, sin una buena guía la probabilidad de fracaso del proyecto aumenta considerablemente.

Desarrollo guiado por el comportamiento

El desarrollo guiado por el comportamiento, o *Behavior-Driven Development (BDD)* es una estrategia de desarrollo que plantea es definir un lenguaje común para el negocio y para los técnicos, y utilizar eso como parte inicial del desarrollo y las pruebas (Toledo, 2017). Una comunicación de buena calidad entre los usuarios y clientes con el equipo de desarrollo es importante para que conseguir los objetivos trazados en el proyecto de desarrollo de software; en muchas ocasiones designan personas como traductores del lenguaje del negocio al lenguaje del desarrollo, lo que llega a generar mucho ruido en la comunicación. BDD propone definir en lenguaje común que todos los involucrados entiendan y se sientan cómodos usándolo.

> BDD cierra la brecha entre la gente de negocios y la gente técnica al:
>
> - Fomentar la colaboración entre roles para construir una comprensión compartida del problema a resolver.
> - Trabajar en iteraciones pequeñas y rápidas para aumentar la retroalimentación y el flujo de valor.
> - Producir documentación del sistema que se coteja automáticamente con el comportamiento del sistema.
>
> Se hace esto enfocando el trabajo colaborativo en ejemplos concretos del mundo real que ilustran cómo se quiere que se comporte el sistema. Se usan esos ejemplos para guiar desde el concepto hasta la implementación, en un proceso de colaboración continua (SmartBear Software, 2022).

Al existir un único lenguaje, los usuarios y el cliente pueden interactuar de mejor manera con el equipo, no solo en las reuniones sino haciendo uso de artefactos de software que les permitan definir lo que necesitan y verificar el avance del proyecto de desarrollo de software. Además, del lenguaje común las iteraciones cortas ayudan al equipo de desarrollo a entender de mejor manera lo que necesitan los usuarios y el cliente. Mientras más automatizado este el proceso de verificación de las funcionalidades del software con los requerimientos, más rápida será la retroalimentación que el equipo recibe de los usuarios.

Terhorst-North (2006) se cuestiona y responde, ¿Cuál debería ser el nombre de una prueba? Fácil, el nombre es una frase que describe el comportamiento de esa parte del sistema. Los usuarios y el cliente son los expertos del comportamiento que necesitan del software, por lo tanto deberían ser capaces de estructurar frases que definan clara y adecuadamente dicho comportamiento.

2.3 DESARROLLO DE SOFTWARE LIBRE

La metodología y las herramientas utilizadas en el desarrollo de software de código abierto están impulsadas por el tipo de actividad de desarrollo que implementa la empresa; las metodologías deben apoyar, guiar y empoderar para entregar la mejor solución posible (Jääskeläinen, 2013). Los equipos de desarrollo de los proyectos de software libre usan diferentes metodologías y herramientas de desarrollo dependiendo del proyecto y de las características del equipo; sin importar la elección la metodología y las herramientas ayudan a conseguir sus objetivos y constituyen una parte importante de la organización del proyecto.

2.3.1. *Open Source Development Model*

Según Haddad and Warner (2011), el *Open Source Development Model* adopta un enfoque diferente, favoreciendo un proceso de desarrollo más fluido caracterizado por una mayor colaboración dentro del equipo, integración y pruebas continuas y una mayor participación del usuario final. Es un proceso ágil de desarrollo de software con alta participación de la comunidad que desarrolla y usa el software, participación de los usuarios finales permite a los desarrolladores retroalimentación para corregir los errores e implementar nuevas funcionalidades.

El *Open Source Development Model* supone que el desarrollo se distribuye entre varios equipos, que trabajan en diferentes ubicaciones, en una estructura fluida que es resistente a las nuevas llegadas o salidas (Haddad and Warner, 2011). Está pensado para gestionar el trabajo de muchas personas a lo largo de mundo, donde es necesario coordinar a todos los participantes para aprovechar de mejor manera el trabajo y recursos disponibles.

> El *Open Source Development Model*, comienza con una idea para un nuevo proyecto, una nueva funcionalidad o capacidad para un componente ya existente. El siguiente paso es proporcionar un diseño para la implementación y luego un prototipo funcional. En el momento en que se ejecuta el software, se lanza como versión de desarrollo, aunque puede contener errores conocidos y desconocidos. El software será probado por la comunidad, la cual discute y proporciona comentarios, informes de errores y correcciones. Los miembros del proyecto y los encargados del mantenimiento registran y toman en cuenta los comentarios para mejorar la implementación y, a continuación, estará disponible una nueva versión de desarrollo. Este ciclo ocurre tantas veces como sea necesario hasta que los miembros del proyecto sientan que la implementación es lo suficientemente estable (Haddad, 2008).

La participación de la comunidad es muy importante dentro del ciclo de vida del proceso; en la figura II.1 se puede ver que no existe una clara diferencia entre los desarrolladores y los usuarios siendo tomados como miembros de la comunidad que pueden participar en todas las decisiones del software.

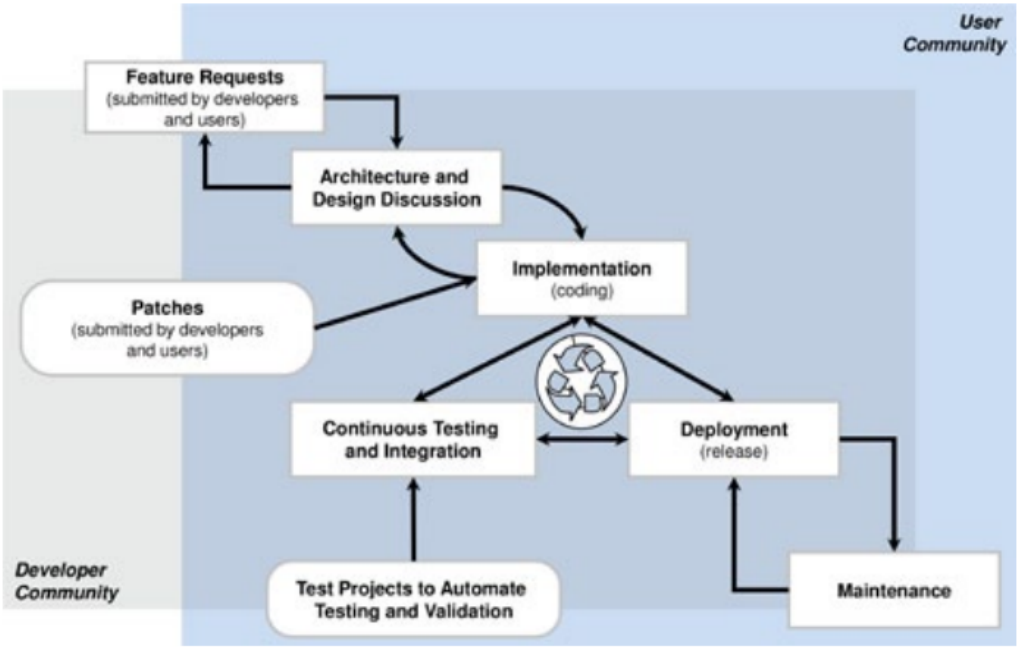

Figura II.1: Ciclo de vida del *Open Source Development Model*
Fuente: Haddad and Warner (2011)

Sus principales características son:

Ciclo de desarrollo entretejido Fomenta el desarrollo de funciones continuo e independiente; esto permite integrar nuevas funciones a medida que están listas, lo que a su vez permite a otros desarrolladores desarrollarlas más rápidamente y producir un producto más competitivo (Haddad and Warner, 2011). Las funcionalidades se libaran a producción tan pronto están terminadas, pueden existir varias personas trabajando en la misma funcionalidad, se integra el código de la primera solicitud que cumpla con los estándares de calidad del proyecto, esto permite generar una sana competencia entre los miembros de la comunidad.

Lanzamiento temprano y con frecuencia Publicar código alfa en la comunidad de desarrollo para su revisión mucho antes de la versión final, da como resultado un desarrollo altamente iterativo y minimiza la cantidad de cambios entre versiones de desarrollo y facilita el diagnóstico de errores (Haddad and Warner, 2011). La mejor manera de evaluar la calidad del producto es cuando los usuarios finales lo pueden usar; muchos miembros de las comunidades de software aceptan instalar versiones alfa para ayudar con el proceso de identificación y corrección de errores, sin importar las habilidades técnicas todo usuario puede ayudar usando el software, reportando los fallos y proponiendo nuevas funcionalidades.

Revisión por pares Cuando una función terminada se considerada para ser integrada otros miembros del proyecto revisan el código y brindan retroalimentación para mejorar la calidad y la funcionalidad; el responsable del proyecto proporciona un nivel de revisión antes de aceptar el código (Haddad and Warner, 2011). El acceso al código fuente permite que muchos ojos revisen y auditen el trabajo realizado; por otro lado permite que los desarrolladores nuevos conozcan el funcionamiento del proyecto y de esta manera puedan contribuir de mejor manera.

2.3.2. Proyectos de aplicaciones libres

Parker (2022) afirma que entre los mejores proyectos libre y de código abierto se pueden observar a los siguientes:

- *LibreOffice*.
- *Brave*.
- *Audacity*.
- Núcleo de *Linux*.

Todos estos proyectos manejan de manera diferente su ciclo de desarrollo.

LibreOffice

LibreOffice sin duda es una de las piezas de software más importante del ecosistema del libre, en vista que es incorporada en muchas distribuciones *Linux*; por lo tanto, del desarrollo de *LibreOffice* depende muchos otros proyectos.

> El modelo de desarrollo por series de publicaciones periódicas ha demostrado producir el software libre de mejor calidad. Los lanzamientos periódicos son aquellos que no se detienen si no se ha terminado una funcionalidad o incluido una corrección de error; en la medida de lo posible, el tiempo es la única métrica. Esto disciplina a los desarrolladores a incluir correcciones a tiempo, proporciona previsibilidad y permite actualizar el producto constantemente (Barrientos, 2022).

Tener versiones nuevas con un calendario claro y periódico ayuda a que otros proyectos puedan tomar sus previsiones para incorporar la mejor versión de *LibreOffice* dentro de sus lanzamientos. Por otro lado, tener plazos fijos permite que los desarrolladores puedan gestionar de mejora manera su tiempo para presentar la mejor versión posible de la funcionalidad en que la están trabajando.

> Hay dos ramas: Nueva (la versión más reciente) y Estable (la versión inmediatamente anterior), las cuales se destinan, respectivamente, a usuarios generales que quieren probar las últimas novedades y a entornos empresariales o de producción que buscan mayor fiabilidad.
>
> Como resultado, los usuarios obtienen una nueva versión principal cada seis meses con una amplia gama de funciones, correcciones y mejoras (Barrientos,

2022).

Un proyecto como *LibreOffice* que usan muchos usuarios necesita satisfacer varias necesidades; por un lado está la necesidad de nuevas funcionalidades que mantienen al proyecto vigente y como un alternativa real ante sus competidores en el mercado; por otro lado, otros usuarios demandan estabilidad y seguridad en las herramientas del proyecto en vista que son parte fundamental de su proceso y una falla podría significa perdidas e inconvenientes. Mantener las dos ramas permite satisfacer las necesidades de los usuarios que demandan nuevas funcionalidades y de los que requieren herramientas sólidas y confiables.

Brave

Brave es navegador basado en *Chromium*, su proceso de desarrollo está compuesto por cuatro fases, como se puede ver en la figura II.2.

Fase 0 La fase 0 del proceso inicia con el aporte de un colaborador del proyecto.

Se cuenta con un sistema de prueba y distribución más diversificado y automatizado para las nuevas versiones del navegador. Las solicitudes de incorporación de cambios a GitHub se prueban y comparan automáticamente con el sistema central por parte de Travis. Luego, estas solicitudes se fusionan con la rama principal y las ramas de liberación de envío. Dado que muchos parches van en un solo lanzamiento, se activa la siguiente etapa solo en ciertos eventos. En general, estos eventos son:

- cuando se tienen suficientes funciones nuevas,
- cuando se lanza una nueva versión de Chromium o
- cuando el equipo de lanzamiento decide impulsar una nueva versión.

Una vez que se ha decido lanzar una nueva versión, comienza el proceso de compilación de la nueva versión (Newman, 2017).

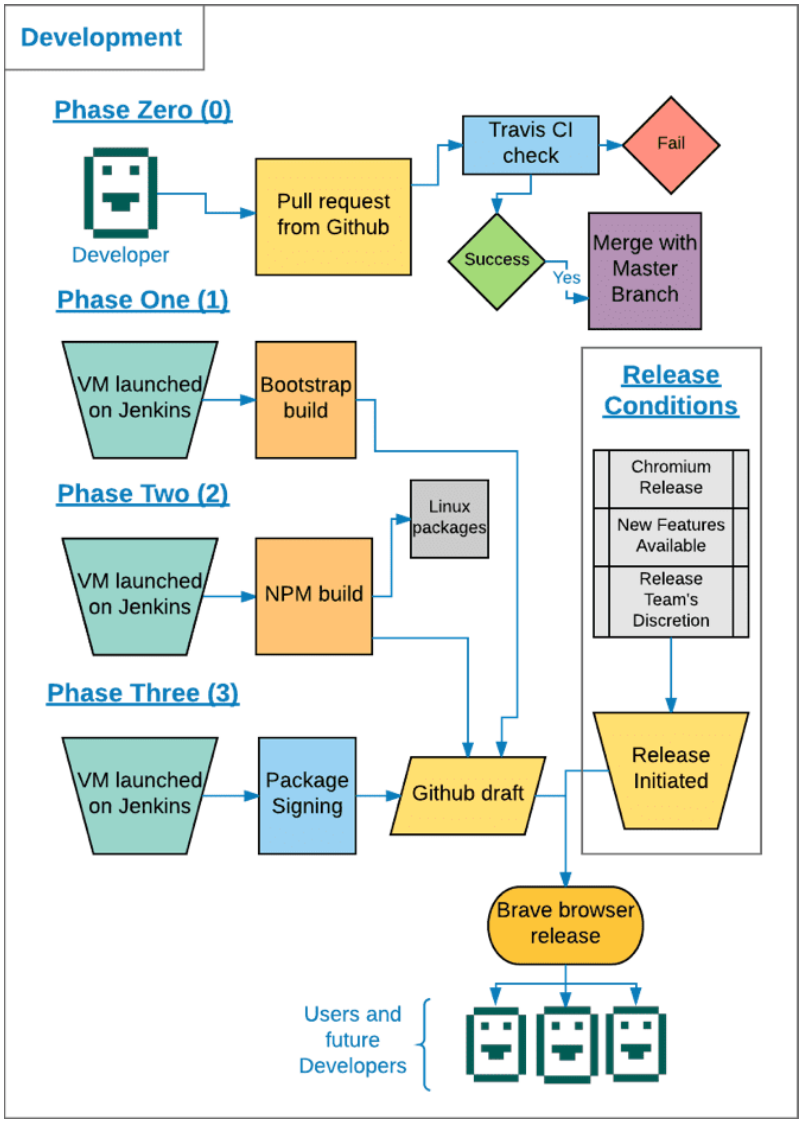

Figura II.2: Mapa de lanzamiento del navegador Brave.
Fuente: Newman (2017).

Al ser un proyecto que depende de otro, *Brave* no puede tener un calendario de lanzamientos independiente, debe tener en cuenta los lanzamientos del proyecto padre para poder incluir y adaptar las nuevas características del mismo. Por otro lado, automatizar el proceso de aceptación de contribuciones ayuda a que el mismo sea más eficiente porque libera a otros miembros del equipo en realizar revisiones de los aportes.

Fase 1 Es importante verificar los efectos de la propuesta en el resto del software, la revisión exhaustiva del funcionamiento de las nuevas piezas de software permiten garantizar la calidad del producto final.

> En esta fase, el desarrollador puede usar herramientas para ver qué objetos se construyen. Esto puede ayudar a comprender qué tan verificables son las compilaciones y acelerar esa fase. Revisar los objetos nuevos permite a los desarrolladores crear de forma independiente el mismo software desde el mismo caché. Después de la primera compilación, se publica un borrador en *GitHub*, donde espera una segunda ronda de pruebas e integración (Newman, 2017).

Una vez que todos los componentes trabajan de manera armoniosa y estable se da por concluida la fase.

Fase 2

> Se toman todas las entradas de la fase uno y se aplica a la composición de la fase dos. Luego se copian los artefactos de la fase uno usando una de las opciones de sincronización en el arranque de la fase uno. Seguidamente usa npm para construir una versión del navegador Brave. Finalmente, se construyen los paquetes de Linux y las versión para Windows y Mac. Una vez que se completa la firma del repositorio, Jenkins publica los (Newman, 2017).

En vista que *Brave* se usa en diferentes plataformas es necesario construir una versión usable para cada una de ellas.

Fase 3

> La fase tres comienza cuando un ingeniero de lanzamiento desbloquea las claves de firma del paquete de Linux. Se mantiene una instancia restringida y bloqueada dedicada al único propósito de firmar paquetes. Solo un pequeño grupo de ingenieros autorizados accede manualmente a este nodo de firma dedicado, o firmante (Newman, 2017).

Finalmente, es necesario que los paquetes y ejecutables se firmen adecuadamente para ser distribuidos y puedan llegar de manera confiable a todos los usuarios.

Audacity

Audacity tiene un conducto para procesar solicitudes de características nuevas, que se expresan en los siguientes tres pasos, que se detallan a continuación:

Adición de solicitudes de funciones

- En primera instancia, las solicitudes de funciones se envían al foro de Audacity.
  - Se pide a los colaboradores que definan claramente la característica, incluidos los beneficios del caso de uso.
  - Las solicitudes de funciones se mantienen abiertas para discusión durante al menos 4 semanas para permitir que los usuarios discutan los pros y los contras, y registren su voto si desean apoyar la función propuesta.
- En la página Solicitudes de funciones.
  - Se pide a los editores de wiki que sigan algunas instrucciones bastante detalladas para editar y votar por funciones para que se puedan mantener en cierto orden.
  - Algunas solicitudes del Foro pueden agregarse inicialmente a Solicitudes de funciones pendientes para su revisión antes de agregarlas a las Solicitudes de funciones en sí.

(Audacity, 2021)

La comunidad juega un papel muy importante en la toma de decisión de las funcionalidades que incluirán en futuras versiones. El debate y la votación se la hace de la manera más pública posible, donde todos los interesados pueden exponer su punto de vista a favor o en contra respecto de las propuestas.

Contribuir a las propuestas

- Un pequeño número de usuarios crea casos de uso que tienen grupos de funciones relacionadas.
    - Los casos de uso harían que *Audacity* se adaptara mejor a alguna aplicación específica. Estos son particularmente útiles para motivar la adición de nuevas características más grandes.
    - A veces, las páginas de propuestas van acompañadas de páginas de notas de antecedentes en las que se comparan las opciones.

(Audacity, 2021)

Agrupar las funcionalidades por caso de uso, es decir funcionalidades necesarias para realizar alguna tarea específica ayuda entender el propósito de las mismas y reduce el riesgo de incorporar funcionalidades que no tendrán un uso real.

Implementación, Pruebas y Liberación

- Los desarrolladores escriben su función con una definición condicional, por lo que la función se puede activar o desactivar.
    - La función está habilitada en versiones experimentales de *Audacity* mucho antes de que esté disponible para los usuarios en general.
    - Las nuevas características se prueban, depuran y refinan.
    - El diseño puede cambiar bastante en esta etapa como resultado de la retroalimentación.
- Después de la discusión, algunas de las funciones en desarrollo se incorporan a la hoja de ruta y están programadas para un número de lanzamiento definitivo de *Audacity*.

(Audacity, 2021)

La disponibilidad de nuevas funcionalidades en las versiones experimentales permite a los usuarios, que así lo requieren, contar con las funcionalidades lo más pronto posible y a los desarrolladores recibir retroalimentación ya sea para corregir errores o para mejorar

la funcionalidad. Las versiones estables cuentan con características que ha sido probadas y refinadas por muchos miembros de la comunidad.

Núcleo de *Linux*

Según The kernel development community (2022), las etapas por las que pasa un aporte o parche son, generalmente:

Diseño. Es donde se establecen los requisitos reales para el parche, y la forma en que se cumplirán esos requisitos. El trabajo de diseño a menudo se realiza sin involucrar a la comunidad; puede ahorrar mucho tiempo de rediseño más adelante (The kernel development community, 2022). Del núcleo depende muchos proyectos, por lo tanto la especificación de requerimientos la realiza un grupo pequeño de ingenieros que velan por el mejor camino a seguir para cumplir con las exigencias de otros proyectos de software libre como son las distribuciones para diferentes tipo de dispositivos.

Revisión temprana. Los parches se publican en la lista de correo correspondiente y los desarrolladores de esa lista responden con cualquier comentario que puedan tener. Este proceso debería mostrar cualquier problema importante con un parche si todo va bien (The kernel development community, 2022). La retroalimentación temprana ayuda a encontrar defectos lo más pronto posible y corregir; además, permite entregar un producto de mejor calidad.

Revisión más amplia. Cuando el parche esté casi listo, debe ser aceptado por un responsable de mantenimiento; el parche aparecerá en el árbol del mantenedor; cuando funciona conduce a una revisión exhaustiva y al descubrimiento de problemas resultantes de la integración del parche (The kernel development community, 2022). Muchas veces una pieza de software puede solucionar adecuadamente el problema para la que fue creada pero puede generar problemas con otros componentes ya existentes; por lo tanto, es necesario una revisión más profunda para identificar los posibles conflictos con el resto del proyecto.

Fusión en la línea principal. Eventualmente, un parche exitoso se fusionará con el re-

positorio principal administrado por Linus Torvalds. Más comentarios o problemas pueden surgir en este momento; es importante que el desarrollador responda a estos y solucione cualquier problema que surja (The kernel development community, 2022). Linus tiene la última palabra al momento de aprobar o rechazar un aporte, es la revisión final que permite al parche unirse definitivamente con el resto del código del núcleo.

Liberación estable. La cantidad de usuarios potencialmente afectados por el parche ahora es grande, por lo que, una vez más, pueden surgir nuevos problemas (The kernel development community, 2022). Después de superar todas las revisiones y realizar todas las enmiendas y mejorar el parche sale finalmente a los usuario finales. La retroalimentación de los usuarios y otros proyectos que usan el núcleo es importante para corregir los defectos y entregar el mejor producto posible.

Mantenimiento a largo plazo. Es posible que un desarrollador se olvide del código después de fusionarlo, ese comportamiento deja una mala impresión en la comunidad de desarrollo; el desarrollador debe continuar asumiendo la responsabilidad del código para que siga siendo útil a largo plazo (The kernel development community, 2022). El trabajo no termina al fusionar el código con el proyecto, por el contrario recién empieza, el compromiso con el proyecto por parte de los desarrolladores los lleva a dar soporte a sus parches para que sigan siendo útiles; este compromiso permite que el núcleo sea usado por tantos proyectos al rededor del mundo.

La comunidad, sin duda, es la pieza fundamental dentro del desarrollo de Software Libre, participa en todas las etapas del ciclo de desarrollo del proyecto. Es la comunidad la que hace posible la existencia del proyecto, dando retroalimentación, aportando con las construcción de artefactos de software, realizando documentos para otros usuarios y haciendo aportes económicos.

CAPÍTULO III

METODOLOGÍA DE INVESTIGACIÓN

3.1 MÉTODOS EMPÍRICOS

Encuesta. Para la caracterización del contexto y diagnóstico de la problemática se realizarán encuestas a equipos muy pequeños de desarrollo de Software Libre.

Entrevista. Para la caracterización del contexto y diagnóstico de la problemática se realizarán entrevistas a los:

- Responsables de equipo.

Estudio documental. Para la caracterización del contexto y diagnóstico de la problemática se realizará el estudio de los siguientes documentos:

- Plan de Implementación de Software Libre y Estándares Abierto.

Para la construcción de marco teórico se realizará el estudio de los siguientes componentes teóricos:

- Desarrollo de software.
- Marco de trabajo.
- Agilidad en el desarrollo de software.
- Manifiesto ágil.
- Marcos de trabajo ágiles.
- Proyectos libres.

Criterio de expertos. Para verificar la viabilidad y validez de la propuesta se aplicará el método Delphi a expertos de desarrollo de software que cumplan funciones de organización de equipos.

3.2 MÉTODOS TEÓRICOS

Analítico y sintético. Se analizarán los principios de la agilidad en el desarrollo de software, los principales marcos de trabajo ágiles que no definan roles específicos en vista que en equipos muy pequeños, de uno a tres miembros, es difícil cumplir con los roles.

Histórico lógico. Se realizará un análisis histórico sobre la evolución de los marcos de trabajo ágiles y como han afectado al desarrollo de software.

3.3 MUESTRA

Encuesta

Población: Desarrolladores que hayan participado en el desarrollo de Software Libre.

Muestra: Probabilista.

Criterios de inclusión: Pertenecer a un equipo formado entre uno y tres miembros.

Entrevista

Población: Responsables de equipo.

Muestra: Probabilista.

Criterios de inclusión: Liderar equipos que hayan participado en proyectos de desarrollo de Software Libre.

3.4 MÉTODOS DE INVESTIGACIÓN

Encuesta. Para la caracterización del contexto y diagnóstico de la problemática se realizarán encuestas a desarrolladores que hay participado en equipos muy pequeños en el desarrollo de Software Libre.

Entrevista. Para la caracterización del contexto y diagnóstico de la problemática se realizarán entrevistas a los responsables de equipos muy pequeños que han desarrollado

Software Libre.

Estudio documental. Para la caracterización del contexto y diagnóstico de la problemática se realizará el estudio de los siguientes documentos:

- Plan de Implementación de Software Libre y Estándares Abiertos.

Para la construcción de marco teórico se realizará el estudio de los siguientes componentes teóricos:

- Desarrollo de software.
- Marco de trabajo.
- Agilidad en el desarrollo de software.
- Manifiesto ágil.
- Marcos de trabajo ágiles.
- Proyectos libres.

Criterio de expertos. Para verificar la viabilidad y validez de la propuesta se aplicará el método Delphi a expertos de desarrollo de software y con conocimiento de Software Libre.

3.5 TIPO DE INVESTIGACIÓN

Esta investigación se sitúa dentro del paradigma del racionalismo crítico, ya que reconoce que el conocimiento científico es provisional y requiere una evaluación y una revisión constante. Las metodologías para el desarrollo de software en un contexto general, y específicamente en el ámbito del Software Libre, deben ser flexibles y estar en permanente evolución para adaptarse a las nuevas tecnologías, las demandas cambiantes del mercado y las peculiaridades de los equipos de desarrollo.

La investigación tecnológica se centra en la aplicación del conocimiento científico y técnico para diseñar y construir soluciones que satisfagan las necesidades humanas y mejoren la calidad de vida de la sociedad en general. La presente investigación se considera de tipo

tecnológica, en vista que se proporcionará un marco de trabajo que permita a los equipos pequeños desarrollar de mejor manera sus actividades dentro de un proyecto de Software Libre.

3.6 UNIVERSO O POBLACIÓN DE ESTUDIO

Está compuesto por profesionales que hayan participando en el desarrollo de Software Libre desde 2019.

3.6.1. Determinación y Elección de la Muestra

Se han tomando personas de forma aleatoria.

3.7 SUJETOS VINCULADOS A LA INVESTIGACIÓN

La Universidad San Francisco Xavier de Chuquisaca cuenta una gran cantidad de carreras en Tecnologías de Información y Comunicación (TIC), como ser Ingeniería de Sistemas orientada a la captura de requerimientos e Ingeniería en Ciencias de la Computación orientada al desarrollo de software. Se constituye en la institución de formación superior que proporciona la mayor cantidad de profesionales en TIC de la ciudad de Sucre.

El *Google Developer Group* de Sucre, es una comunidad que aglutina a profesionales y estudiantes en TIC, que permite el aprendizaje, cooperación y fortalecimiento de conocimientos y habilidad en las TIC tanto en el desarrollo de software como en su uso, haciendo énfasis a herramientas y tecnología desarrolladas por Google, muchas de las cuales tienen licencias libres.

Los profesionales de las TIC de Sucre que desarrollan Software Libre tanto en empresas públicas como privadas. Además, promueven y difunden el uso del Software Libre en diferentes conferencias y ferias.

3.8 FUENTES Y DISEÑO DE LOS INSTRUMENTOS DE RELEVAMIENTO DE INFORMACIÓN

Se tendrán siguientes fuentes de investigación:

Primarias

- Encuesta a los desarrolladores.
- Entrevista a responsables de equipo.

Secundarias

- Revisión bibliográfica de las principales metodologías y marcos de trabajo relacionadas con el desarrollo de software.

3.8.1. Diseño de los Instrumentos de Relevamiento de Información

Se han diseñado los siguientes instrumentos:

- Encuesta a desarrolladores.
- Guía de entrevista para responsables de equipo.
- Consulta a expertos.

Encuesta a desarrolladores

El objetivo de la presente encuesta es caracterizar el proceso de desarrollo de software libre.

Los resultados serán empleados como parte del trabajo de investigación de una tesis de maestría.

Por favor responda con la mayor sinceridad posible. Toda la información será recopilada de forma anónima.

1. Mencione metodologías o marcos de trabajo para el desarrollo de software que ha estudiado o ha trabajado con ellas.

    1. _____

    2. _____

    3. _____

2. ¿Cuál es la metodología o marco de trabajo que usa en su trabajo diario?

    _____

3. ¿Existen alguna característica que destaque como muy positiva de la metodología o marco de trabajo que usa en su trabajo diario?

    _____

4. ¿Existen alguna característica que destaque como muy negativa de la metodología o marco de trabajo que usa en su trabajo diario?

    _____

5. ¿Qué roles existen en el equipo? (Un rol lo puede desempeñar más de una persona)

    _____

    _____

    _____

6. ¿Cuántas personas conforman el equipo?

   ◯ 1 ◯ 2 ◯ 3

7. ¿Cuántas horas a la semana dedica a la planificación?

   ---

8. ¿Cuántas horas a la semana dedica a la programación?

   ---

9. ¿Cuántas horas a la semana dedica a las pruebas?

   ---

10. ¿Cuántas horas a la semana dedica a interactuar con el cliente y los usuarios?

    ---

11. ¿Qué herramientas usa para las diferentes etapas del desarrollo?

Guía de entrevista para responsables de equipo

Organización del equipo

1. ¿Cuántos miembros tiene un equipo de desarrollo?
2. ¿Cómo es su proceso de desarrollo de software? ¿Qué metodología de trabajo usan?
3. ¿Qué roles tienen?
4. ¿Cómo distribuyen su tiempo?
5. ¿Qué problemas tienen en el proceso de desarrollo actual?

Artefactos

1. ¿Qué artefactos o diagramas generan?
2. ¿Quiénes usan esos artefactos?

Herramientas

1. ¿Qué herramientas usan para el desarrollo? (IDE, Control de versiones)
2. ¿Qué herramientas usan para la comunicación?

Consulta a expertos

La investigación titulada "Marco de trabajo ágil para el desarrollo de Software Libre" tiene como objetivo diseñar un marco de trabajo para mejorar el proceso de desarrollo de Software Libre de equipos muy pequeños.

En vista de su trayectoria profesional en el desarrollo de software y del Software Libre por favor revise la propuesta, valore y responda de manera objetiva los siguientes aspectos.

Datos personales

- Años de experiencia profesional: _____
- Titulo Académico: _____
- Grado Académico: _____
- Lugar de trabajo: _____
- Cargo: _____

Grado de conocimiento

Muy bajo	◯	◯	◯	◯	◯	◯	◯	◯	◯	◯	Muy alto
	1	2	3	4	5	6	7	8	9	10	

Fuentes de argumentación o fundamentación

	Alto	Medio	Bajo
Análisis teóricos realizados por usted	◯	◯	◯
Su experiencia obtenida	◯	◯	◯
Trabajos de autores nacionales	◯	◯	◯
Trabajos de autores extranjeros	◯	◯	◯
Su conocimiento del estado del problema en el extranjero	◯	◯	◯
Su intuición	◯	◯	◯

Aspecto	**Muy adecuado**	**Bastante adecuado**	**Adecuado**	**Poco adecuado**	**No adecuado**
Los profesionales del medio pueden emplear el marco de trabajo.	◯	◯	◯	◯	◯
Se puede emplear el marco de trabajo con los recursos disponibles.	◯	◯	◯	◯	◯
Los aspectos de la comunidad del marco de trabajo son pertinentes.	◯	◯	◯	◯	◯
Las actividades del marco de trabajo son pertinentes.	◯	◯	◯	◯	◯
Los artefactos del marco de trabajo son pertinentes.	◯	◯	◯	◯	◯

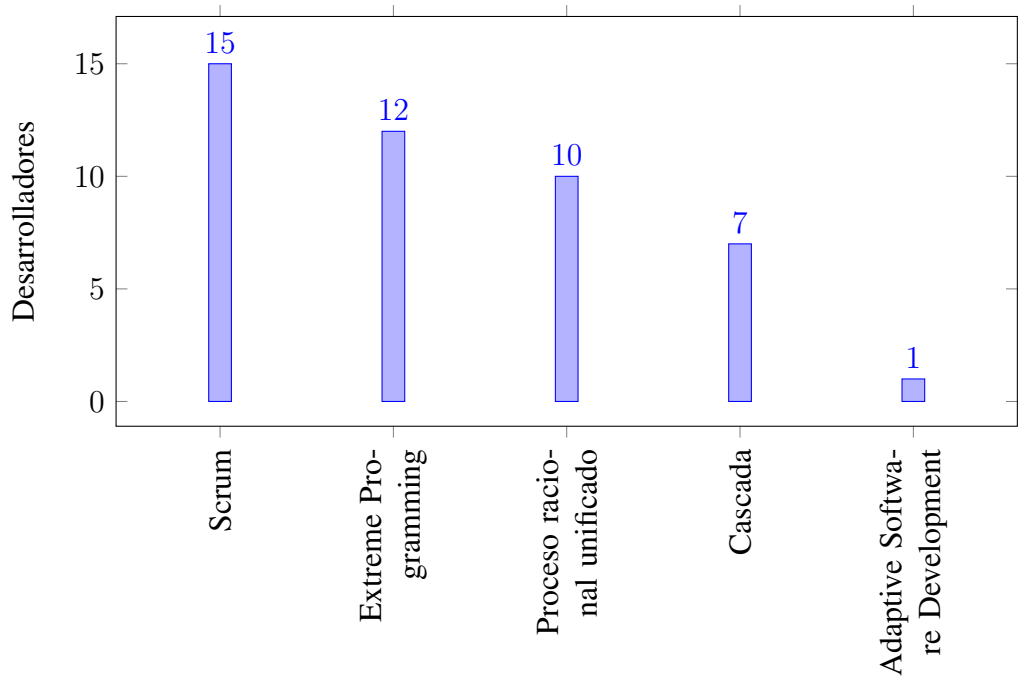

Figura III.1: Conocimiento de metodologías o marcos de trabajo para el desarrollo de software
Fuente: Elaboración Propia.

3.9 PROCESAMIENTO Y ANÁLISIS DE LA INFORMACIÓN

3.9.1. Análisis de de la encuesta a desarrolladores

Encuesta de desarrolladores

Se realizaron encuestas a 17 desarrolladores de Sucre, todos con experiencia en desarrollo de software en equipos muy pequeños. Las metodologías o marcos de trabajo más conocidos por los desarrolladores son ágiles, Scrum y Extreme Programming representan el $60\,\%$ de las respuesta obtenidas, como se puede ver en la figura III.1. Las metodologías tradicionales ocupan el segundo lugar, el Proceso racional unificado y la Cascada ocupan el $38\,\%$. Todas las respuestas requieren de muchas personas para implementarlas de manera adecuada en vista que incluyen roles bien definidos.

Sin duda Scrum no solo es el marco de trabajo más conocido por los desarrolladores, es el más usado, como se puede ver en la figura III.2. Scrum tiene el $65\,\%$ en la preferencia

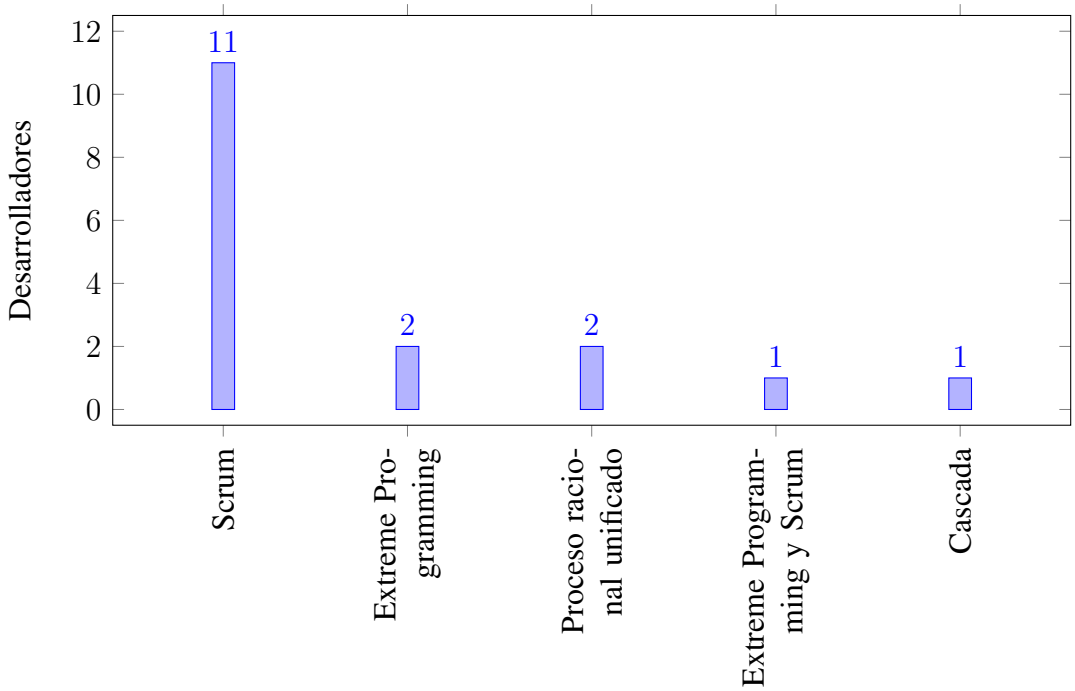

Figura III.2: Uso de metodologías o marcos de trabajo para el desarrollo de software
Fuente: Elaboración Propia.

de uso de los equipos de desarrollo. Si embargo, Scrum define tres roles por lo tanto si el equipo es muy pequeño requiere que los miembros del mismo tenga que desempañar más de un rol, lo cual no es recomendable en vista que puede generar conflictos de intereses según los objetivos que tiene cada rol.

Los equipos se organizar de maneras diferentes, los encuestados han mencionado 55 roles; siendo el rol de programador el único que se ha nombrado por todos los participantes, como se puede ver en la figura III.3. *Product Owner* y Diseñador y administrador de base de datos son los siguientes roles con mayor frecuencia, lo cual muestra que los equipos ponen énfasis en la captura de requerimientos y en el diseño adecuado de la base datos. Llama la atención que si bien Scrum es la más usado solo el $9\,\%$ de las respuestas representan al Scrum Master.

El tamaño más común de los equipos muy pequeños es tres que representa el $52{,}9\,\%$, como se puede ver en la figura III.4. Si bien la mayoría son equipos de tres miembros, aun existen equipos con tamaños más pequeños. Mientras más pequeños más difícil aplicar correctamente las metodologías y marcos de trabajo para el desarrollo de software, en vis-

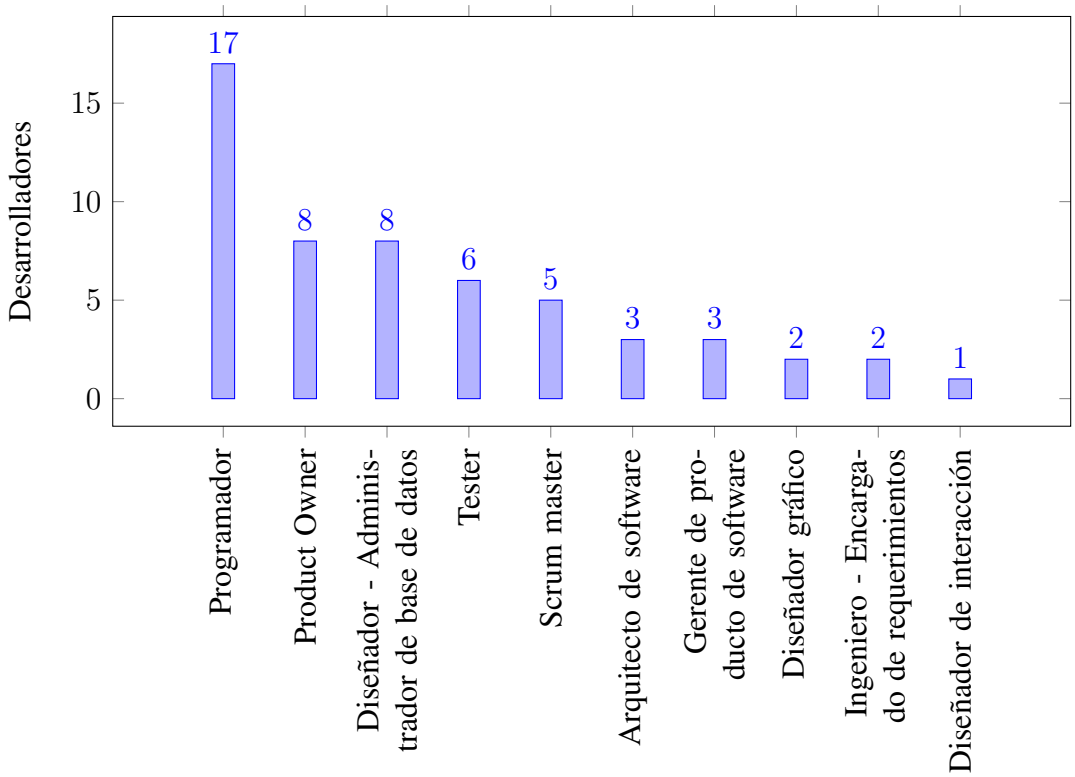

Figura III.3: Roles existentes en el equipos de desarrollo
Fuente: Elaboración Propia.

ta que es necesario que todos sus miembros desarrollen varios roles, generando confusión en las responsabilidades, actividades y artefactos que debe producir cada miembro.

La planificación es una actividad que se realiza poco en los equipos pequeños, en el mejor caso ocupa el $30\,\%$ del tiempo semanal de desarrollo y en el pero no se realiza, como se puede ver en la figura III.5. El $12,5\,\%$ y el $16,67\,\%$ representan los tiempos con mayor frecuencia, cual representa casi una jornada de trabajo.

Por otro lado, la programación es la actividad que más realizan los equipos muy pequeños, usando en algunos casos el $100\,\%$ de su tiempo para esta actividad, el $50\,\%$ es el tiempo más bajo destinado a la programación, como se puede ver en la figura III.6. Ciertamente, en equipos muy pequeños, la prioridad es entregar software lo más pronto posible y por eso emplean la mayor cantidad de tiempo y esfuerzo a la programación.

La calidad de software es un aspecto poco relevante en el trabajo de los equipos en vista que emplean poco tiempo para realizar pruebas que garanticen el cumplimiento de las funcionalidades requeridas, como se puede ver en la figura III.7. En el peor caso no se

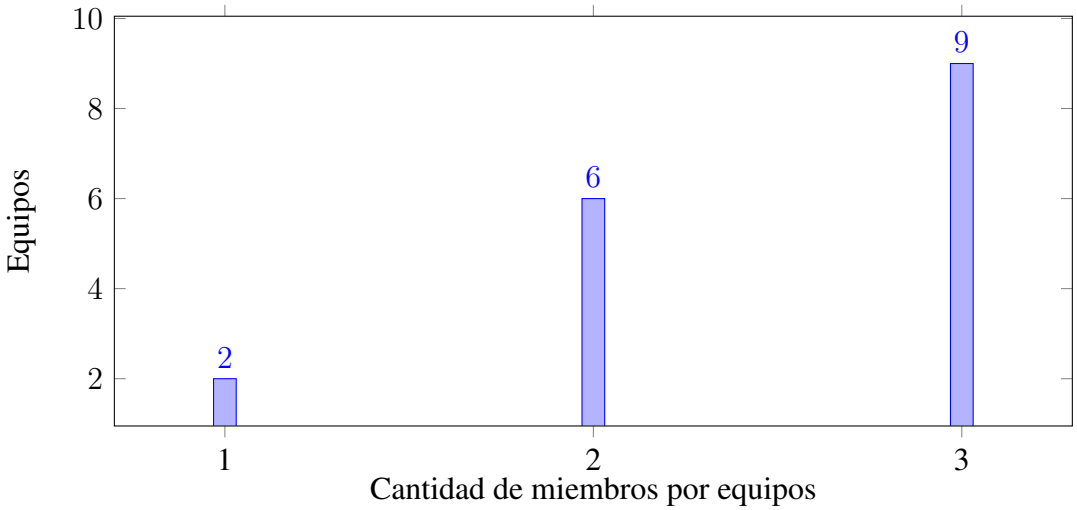

Figura III.4: Tamaño de los equipos de desarrollo
Fuente: Elaboración Propia.

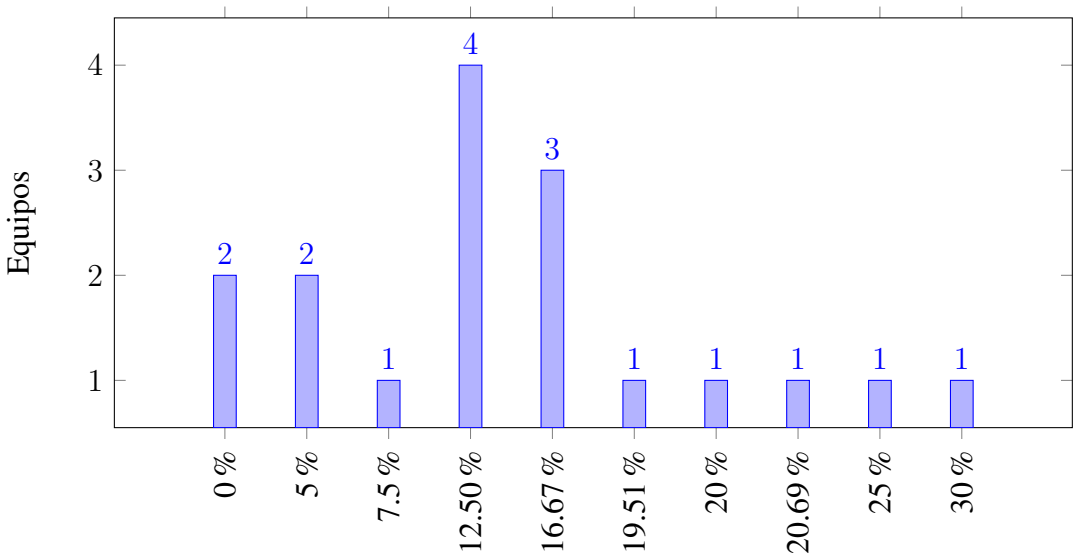

Figura III.5: Tiempo semanal dedicado a la planificación
Fuente: Elaboración Propia.

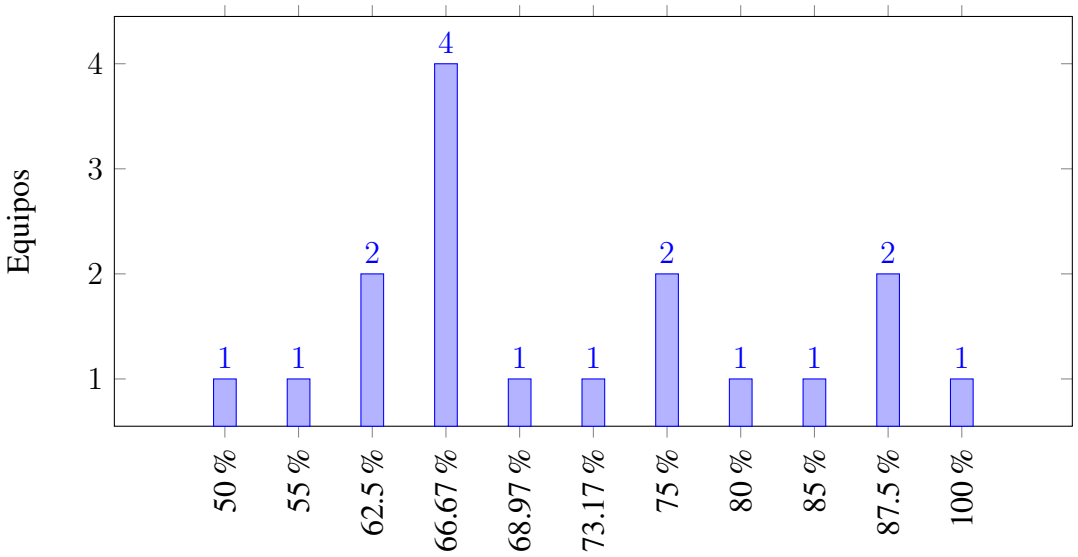

Figura III.6: Tiempo semanal dedicado a la programación
Fuente: Elaboración Propia.

destina tiempo alguno para realizar pruebas, los equipos usan como máximo un $20\,\%$ de su tiempo para destinarlo a las pruebas y la calidad del software. El $5\,\%$ y $10\,\%$ representan los tiempos con mayor frecuencia, aspecto que implica entregar software sin el debido proceso de aseguramiento de la calidad.

La interacción con el cliente es la actividad que menos tiempo destinan los equipos muy pequeños, en el mejor escenario emplean el $12{,}5\,\%$ del tiempo semanal, como se puede ver en la figura III.8. Si bien el desarrollo ágil en general y Scrum en particular definen un fuerte interacción con el cliente los equipos interactúan muy poco, esto se puede evidenciar en que $0\,\%$ y $2{,}5\,\%$ son los tiempo con mayor frecuencia.

3.9.2. Entrevista a responsable de equipo

Se realizan tres entrevista a profesionales que están al frente de equipos de desarrollo, que han participado en Desarrollo de software Libre para los sectores privado y público.

Los equipos no usan ninguna metodología o marco de trabajo para el desarrollo de software en vista que ninguna de las opciones conocidas por los miembros de equipo les ayuda efectivamente a cumplir con sus objetivos, han intentado utilizar Scrum con algunas modificaciones pero no han tenido buenos resultados. Los responsables saben que es

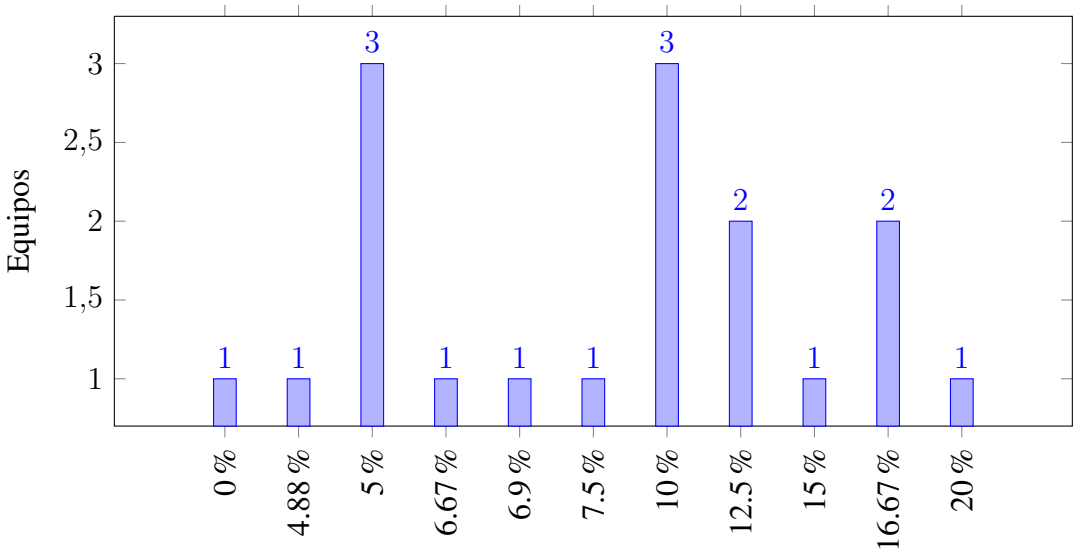

Figura III.7: Tiempo semanal dedicado a las pruebas
Fuente: Elaboración Propia.

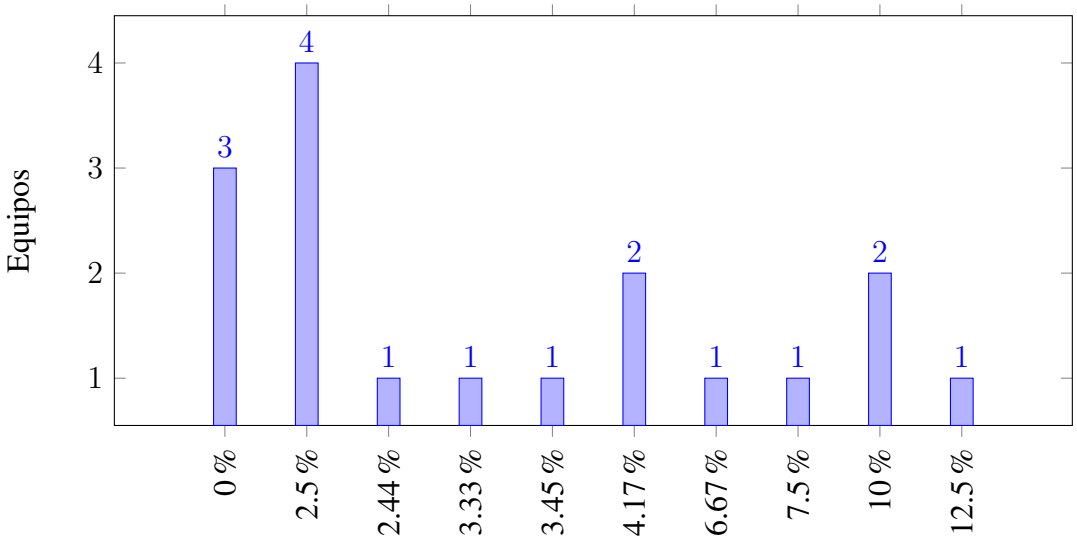

Figura III.8: Tiempo semanal dedicado a interactuar con el cliente y los usuarios
Fuente: Elaboración Propia.

importante aplicar la Ingeniería de Software por lo cual han definido actividad propias como ser reuniones frecuentes con el equipo para sincronizar el avance.

La actividad más importante es la programación, destinan el mayor tiempo y esfuerzo a construir funcionalidades del software. Por otro lado, las pruebas son las que menos tiempo tienen asignado en algunos casos se realizan cuando el producto se encuentra en producción, las pruebas solo se realizan si hay tiempo. Además, el diseño de base de datos y la interfaz de usuario son actividad a las cuales ponen mucho cuidado, en algunos casos con personas encargadas para estos fines en todos los proyectos que se desarrollan.

El artefacto más utilizado es el código fuente. Por otro lado se emplean artefactos que permitan determinar cómo se deben procesar los datos para convertirlos en la información, para ello se usan diagramas de flujo y de secuencia.

Sobre las herramientas utilizadas, los equipos emplean mensajería instantánea para comunicar entre ellos y con los clientes, además hacen uso de las videollamadas para capturar requerimientos como para presentar avances. Los equipos usan sistemas de control de versiones para organizar y manipular el código fuente del proyecto.

3.9.3. Aplicación el método Delphi para la consulta a expertos

Para validar propuesta se seleccionaron a seis expertos, que están altamente relacionados al desarrollo de software, todos ellos con experiencia en proyectos de Software Libre, las características de los expertos se puede ver en el anexo A.

Se definieron cinco categorías, para realizar la consulta a los expertos:

- Muy adecuado (MA).
- Bastante adecuado (BA).
- Adecuado (A).
- Poco adecuado (PA).
- No adecuado (NA).

Se definieron los siguientes aspectos para validar la propuesta:

Tabla III.1: Grado de conocimiento

Experto	1	2	3	4	5	6	7	8	9	10
1								✓		
2									✓	
3								✓		
4							✓			
5							✓			
6							✓			

Fuente: Elaboración propia.

- Los profesionales del medio pueden emplear el marco de trabajo.
- Se puede emplear el marco de trabajo con los recursos disponibles.
- Los aspectos de la comunidad del marco de trabajo son pertinentes.
- Las actividades del marco de trabajo son pertinentes.
- Los artefactos del marco de trabajo son pertinentes.

El instrumento utilizado para recoger la opinión de los expertos en encuentra la subsección 3.8.1, cuando los expertos dejaron su opinión sobre la propuesta se procedió a realizar el trabajo estadístico del Método Delphi.

Con las respuestas de los expertos de la tabla III.1 se procedió a calcular el Coeficiente de conocimiento:

Para el experto 1 $K_c = 8 \times 0{,}1 = 0{,}8$

Para el experto 2 $K_c = 9 \times 0{,}1 = 0{,}9$

Para el experto 3 $K_c = 8 \times 0{,}1 = 0{,}8$

Para el experto 4 $K_c = 7 \times 0{,}1 = 0{,}7$

Para el experto 5 $K_c = 7 \times 0{,}1 = 0{,}7$

Para el experto 6 $K_c = 7 \times 0{,}1 = 0{,}7$

Con las respuestas de los expertos de la tabla III.2 se procedió a calcular el Coeficiente de argumentación:

Para el experto 1 $K_a = 0,3 + 0,4 + 0,05 + 0,05 + 0,05 + 0,05 = 0,9$

Tabla III.2: Fuentes de argumentación

Fuente de argumentación	Alto	Medio	Bajo
Análisis teóricos realizados por usted	1, 3, 6	2, 4	5
Su experiencia obtenida	2, 3, 4, 5	1, 6	
Trabajos de autores nacionales	4, 5	1, 3, 6	2
Trabajos de autores extranjeros	2, 3	1, 4	5, 6
Su conocimiento del estado del problema en el extranjero	2, 3, 4	1	5, 6
Su intuición	1, 2, 3, 4, 6	5	

Fuente: Elaboración propia.

Para el experto 2 $K_a = 0,2 + 0,5 + 0,05 + 0,05 + 0,05 + 0,05 = 0,9$

Para el experto 3 $K_a = 0,3 + 0,5 + 0,05 + 0,05 + 0,05 + 0,05 = 1$

Para el experto 4 $K_a = 0,2 + 0,5 + 0,05 + 0,05 + 0,05 + 0,05 = 0,9$

Para el experto 5 $K_a = 0,1 + 0,5 + 0,05 + 0,05 + 0,05 + 0,05 = 0,8$

Para el experto 6 $K_a = 0,3 + 0,4 + 0,05 + 0,05 + 0,05 + 0,05 = 0,9$

A continuación, con el Coeficiente de conocimiento y el Coeficiente de argumentación se calculó el Coeficiente de competencia:

Para el experto 1 $K = 0,5 \times (0,8 + 0,9) = 0,85 \approx 0,9$

Para el experto 2 $K = 0,5 \times (0,9 + 0,9) = 0,9 \approx 0,9$

Para el experto 3 $K = 0,5 \times (0,8 + 1) = 0,9 \approx 0,9$

Para el experto 4 $K = 0,5 \times (0,7 + 0,9) = 0,8 \approx 0,8$

Para el experto 5 $K = 0,5 \times (0,7 + 0,8) = 0,75 \approx 0,8$

Para el experto 6 $K = 0,5 \times (0,7 + 0,9) = 0,8 \approx 0,8$

El Coeficiente de competencia del grupo de expertos se puede ver en la tabla III.3, donde se puede observar que todos los expertos tiene uno Coeficiente de competencia Alto, por lo tanto el grupo de expertos puede evaluar de manera adecuada la propuesta.

El expertos evaluaron los diferentes aspectos de la propuesta según las categorías definidas para la evaluación, el resultado se puede ver en la tabla III.4.

Tabla III.3: Coeficiente de competencia

Experto	K_c	K_a	K	Resultado final	Competencia
1	0,8	0,9	0,85	0,9	Alto
2	0,9	0,9	0,9	0,9	Alto
3	0,8	1	0,9	0,9	Alto
4	0,7	0,9	0,8	0,8	Alto
5	0,7	0,8	0,75	0,8	Alto
6	0,7	0,9	0,8	0,8	Alto
Promedio			$0,83$	$0,8$	**Alto**

Fuente: Elaboración propia.

Tabla III.4: Frecuencias de los aspectos y las categorías de evaluación

	Aspectos	MA	BA	A	PA	NA
1	Los profesionales del medio pueden emplear el marco de trabajo.	4	2	0	0	0
2	Se puede emplear el marco de trabajo con los recursos disponibles.	6	0	0	0	0
3	Los aspectos de la comunidad del marco de trabajo son pertinentes.	4	2	0	0	0
4	Las actividades del marco de trabajo son pertinentes.	6	0	0	0	0
5	Los artefactos del marco de trabajo son pertinentes.	6	0	0	0	0

Fuente: Elaboración propia.

Tabla III.5: Frecuencias acumuladas de los aspectos y las categorías de evaluación

	Aspectos	MA	BA	A	PA	NA
1	Los profesionales del medio pueden emplear el marco de trabajo.	4	6	6	6	6
2	Se puede emplear el marco de trabajo con los recursos disponibles.	6	6	6	6	6
3	Los aspectos de la comunidad del marco de trabajo son pertinentes.	4	6	6	6	6
4	Las actividades del marco de trabajo son pertinentes.	6	6	6	6	6
5	Los artefactos del marco de trabajo son pertinentes.	6	6	6	6	6

Fuente: Elaboración propia.

Tabla III.6: Frecuencias acumuladas de los aspectos y las categorías de evaluación finales

	Aspectos	MA	BA	A	PA
1	Los profesionales del medio pueden emplear el marco de trabajo.	0.667	1	1	1
2	Se puede emplear el marco de trabajo con los recursos disponibles.	1	1	1	1
3	Los aspectos de la comunidad del marco de trabajo son pertinentes.	0.667	1	1	1
4	Las actividades del marco de trabajo son pertinentes.	1	1	1	1
5	Los artefactos del marco de trabajo son pertinentes.	1	1	1	1

Fuente: Elaboración propia.

A continuación, se procedió a calcular las frecuencias acumuladas para cada aspecto según las categorías de evaluación, el resultado se puede ver en la tabla III.5.

Siguiendo el procedimiento, se elimina la última columna, se puede ver el resultado en la tabla III.6.

El siguiente paso, consiste en aplicar la Distribución Normal Estándar Inversa, el resultado se puede ver en la tabla III.7.´

Para determinar los puntos de corte es necesario sumar y promedia cada una de las categorías de evaluación, el resultado se puede ver en la tabla III.8.

Con los puntos de corte se identifican las fronteras de las categorías de evaluación, el resultado se puede ver en la tabla III.9, donde se puede observar que los expertos han

Tabla III.7: Distribución Normal Estándar Inversa

	Aspectos	MA	BA	A	PA
1	Los profesionales del medio pueden emplear el marco de trabajo.	0,43	3,9	3,9	3,9
2	Se puede emplear el marco de trabajo con los recursos disponibles.	3,9	3,9	3,9	3,9
3	Los aspectos de la comunidad del marco de trabajo son pertinentes.	0,43	3,9	3,9	3,9
4	Las actividades del marco de trabajo son pertinentes.	3,9	3,9	3,9	3,9
5	Los artefactos del marco de trabajo son pertinentes.	3,9	3,9	3,9	3,9

Fuente: Elaboración propia.

Tabla III.8: Puntos de corte

	Aspectos	MA	BA	A	PA
1	Los profesionales del medio pueden emplear el marco de trabajo.	0,43	3,9	3,9	3,9
2	Se puede emplear el marco de trabajo con los recursos disponibles.	3,9	3,9	3,9	3,9
3	Los aspectos de la comunidad del marco de trabajo son pertinentes.	0,43	3,9	3,9	3,9
4	Las actividades del marco de trabajo son pertinentes.	3,9	3,9	3,9	3,9
5	Los artefactos del marco de trabajo son pertinentes.	3,9	3,9	3,9	3,9
	Suma	12,561	19,5	19,5	19,5
	Puntos de corte	2,51	3,9	3,9	3,9

Fuente: Elaboración propia.

Tabla III.9: Fronteras de las categorías de evaluación

Frontera	Categoría de evaluación
2.51	Muy adecuado
3.9	Bastante adecuado
3.9	Adecuado
3.9	Poco adecuado
3.9	No adecuado

Fuente: Elaboración propia.

evaluado positivamente todos los aspectos en vista que solo existen dos categorías: Muy adecuado y Bastante adecuado.

Para establecer a qué categoría pertenece cada aspecto es necesario calcular N-P, el resultado se puede ver en la tabla III.10.

Tabla III.10: N-P

	Aspectos	MA	BA	A	PA	Suma	Promedio	N-P
1	Los profesionales del medio pueden emplear el marco de trabajo.	0,43	3,9	3,9	3,9	12,131	3,033	-1.256
2	Se puede emplear el marco de trabajo con los recursos disponibles.	3,9	3,9	3,9	3,9	15,6	3,9	-2.123

Continua en la página siguiente.

	Aspectos	MA	BA	A	PA	Suma	Promedio	N-P
3	Los aspectos de la comunidad del marco de trabajo son pertinentes.	0,43	3,9	3,9	3,9	12,131	3,033	-1.256
4	Las actividades del marco de trabajo son pertinentes.	3,9	3,9	3,9	3,9	15,6	3,9	-2.123
5	Los artefactos del marco de trabajo son pertinentes.	3,9	3,9	3,9	3,9	15,6	3,9	-2.123
	Suma	12,561	19,5	19,5	19,5	71,062		
	Puntos de corte	2,51	3,9	3,9	3,9			

Fuente: Elaboración propia.

Los expertos han clasificado a los diferentes aspectos consultados, como se puede ver en la tabla III.11. Todos los aspectos ha sido incluidos en la categoría: Muy adecuado, por lo tanto, se ha llegado al consenso y se concluye la revisión de la propuesta.

Tabla III.11: Clasificación de aspectos

	Aspecto	Categoría
1	Los profesionales del medio pueden emplear el marco de trabajo.	Muy adecuado
2	Se puede emplear el marco de trabajo con los recursos disponibles.	Muy adecuado
3	Los aspectos de la comunidad del marco de trabajo son pertinentes.	Muy adecuado
4	Las actividades del marco de trabajo son pertinentes.	Muy adecuado
5	Los artefactos del marco de trabajo son pertinentes.	Muy adecuado

Fuente: Elaboración propia.

CAPÍTULO IV

RESULTADOS, CONCLUSIONES Y RECOMENDACIONES DE LA INVESTIGACIÓN

4.1 APORTE CIENTÍFICO

Esta marco de trabajo para desarrollo de software se basa en principios ágiles y del Software Libre para ofrecer un enfoque colaborativo y eficiente para crear software de alta calidad que satisfaga las necesidades de los usuarios y la comunidad en general. Su aplicación puede contribuir significativamente al éxito de proyectos de desarrollo de Software Libre de equipos muy pequeños.

4.2 RESULTADOS DE LA INVESTIGACIÓN

A partir de la revisión teórica de diferentes metodologías y marcos de trabajo ágiles se han identificado opciones que no definen roles, lo cual permite que los equipos muy pequeños puedan utilizar de manera efectiva los mismos; pero ninguno está orientado al desarrollo de Software.

Cada comunidad de Software Libre tiene tus propias reglas y normas para su ciclo de desarrollo, es importante conocer, entender y respetar estas normas para poder aportar al software, es necesario convertirse en un miembro de la comunidad con los derechos y obligaciones que esto implica.

Como se puede ver en la figura IV.1 la programación es la actividad que más tiempo ocupada en la semana de trabajo de los equipos muy pequeños, representa el $72\,\%$ del tiempo. Por otro lado, la actividad que menos tiempo es la interacción con los clientes, con un $4\,\%$ si bien los equipos dicen usan metodologías y marcos de trabajo ágiles en la practica no lo hacen en vista que la colaboración con el cliente es un elemento fundamental del desarrollo ágil. La calidad del producto final no tiene el tiempo necesario para las pruebas con

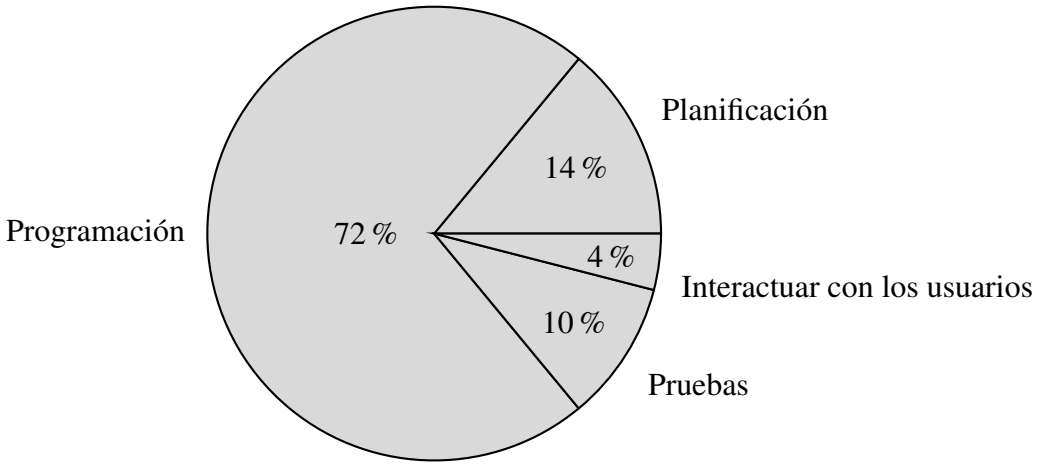

Figura IV.1: Distribución del tiempo semanal
Fuente: Elaboración propia.

apenas el $10\,\%$; en un tiempo tan reducido no es posible asegurar la calidad del software que se entrega.

En equipos muy pequeños no es posible definir muchos roles, en vista de que el tamaño de los mismos hace imposible ejercer adecuadamente todas las responsabilidad que implica el rol, los miembros del equipo son responsables de todas las actividades del desarrollo. Se han definido actividades que mejoran la probabilidad de éxito en el desarrollo de software, haciendo hincapié en la captura de requerimientos en historias de usuarios bien detalladas y en las pruebas unitarias y de aceptación automatizas para garantizar que el producto resultante sea de alta calidad y satisfaga las necesidades de los miembros de la comunidad en particular y de los usuarios en general.

4.3 CONCLUSIONES GENERALES DE LA INVESTIGACIÓN

Se han sistematizado referentes teóricos sobre el desarrollo ágil de software que no definen roles, las metodologías estudiadas han sido el Desarrollo guiado por pruebas de software, el Desarrollo guiado por pruebas de aceptación y el Desarrollo guiado por el comportamiento, en vista que permiten desarrollar software de calidad sin importar el número de personas que componen el equipo.

Los desarrolladores de equipos muy pequeños conocen diferentes metodologías y marcos

de trabajo para el desarrollo de software, la gran mayoría sostiene que usan Scrum con modificaciones pero en la práctica lo más importante es la programación para entregar software lo más pronto posible sin importar la calidad del producto.

Se han definido actividades y artefactos que permitirán a los equipos muy pequeños mejorar sus probabilidades de éxito cuando tengan que desarrollar Software Libre, dando hincapié a conocer la comunidad para poder integrarse a la misma o dando las pautas para formar una comunidad nueva.

A partir de las consulta a expertos se ha podido determinar que la propuesta se puede aplicar por los profesionales del medio con los recursos y presupuestos con los que disponen. Además, los expertos consideran pertinentes todos los elementos de la propuesta.

Finalmente, se ha diseñado un marco de trabajo ágil para equipos muy pequeños, que permite organizarlos para que pueda cumplir de mejor manera con sus compromisos y objetivos; los desarrolladores de Sucre pueden utilizalo con los conocimientos y recursos que tienen disponibles.

4.4 RECOMENDACIONES DE LA INVESTIGACIÓN

La comunidad es el factor más importante a la hora de desarrollar software, muchos proyectos terminan cerrándose por falta de apoyo de las personas. El Software Libre no son las licencias ni el código, son las personas como interactúan entre sí. Se recomienda estudiar las causas para que una comunidad de Software Libre crezca y se mantenga vigente en el tiempo, generando software de calidad para el beneficio de los miembros y del público en general.

El aseguramiento de la calidad es una actividad muy importante dentro del desarrollo de software, se recomienda explorar estrategias para mejorar la manera y el tiempo empleado en escribir y ejecutar pruebas unitarias, de aceptación, de rendimiento, de usabilidad, de accesibilidad, de interfaz de programación de aplicaciones y de interfaz de usuario.

CAPÍTULO V

PROPUESTA DE MEJORAMIENTO

5.1 OBJETIVOS

- Definir las actividades del proceso de desarrollo de software libre.
- Especificar los artefactos necesarios para las actividades del proceso de desarrollo de software libre.

5.2 ALCANCES

Los aspectos de la ingeniería de software que se tomarán en cuenta son solo aquellos referidos al desarrollo ágil. El producto a desarrollarse es Software Libre. Las actividades y artefactos son para equipos de desarrollo pequeños de uno a tres miembros. Los equipos estudiados son de la ciudad de Sucre.

5.3 RESUMEN EJECUTIVO

La comunidad es el aspecto más importante en un proyecto de Software Libre, es importante conocer diferentes aspectos de la misma. Es necesario saber las herramientas de comunicación asíncronas y síncronas y el protocolo de comunicación. Por otro lado, se requiere comprender todos los elementos técnicos y herramientas relacionadas al desarrollo de software. Además, es imprescindible conocer las versiones disponibles, el mantenimiento que reciben y el tiempo de vigencia. Finalmente, se demanda conocer y entender la licencia y todos los aspectos legales relacionados a la propiedad intelectual.

Las actividades para convertir una necesidad en software se enmarcan en tres momentos:

Implementación: Desarrollar el software con calidad que satisface un requerimiento. Usando tres pasos: Escribir prueba unitaria, Escribir código y Refactorización.

Especificación: Desarrollar el software con calidad que satisface el comportamiento requerido. Usando tres pasos: Especificar requerimiento, Implementación y Refactorización.

Exploración: Entregar a la comunidad software con calidad que cumple con las expectativas de los miembros. Usando tres pasos: Definir requerimiento, Especificación y Lanzamiento.

Los artefactos de software que son necesarios producir para cumplir adecuadamente con las actividades son:

Historias de usuario: Recogen y modelan los requerimientos de los usuarios.

Código fuente: Define las instrucciones que compone el software y las pruebas que garantizan la calidad; permite mostrar objetivamente el avance del proyecto.

Tablero Kanban: Visualiza el estado y prioridad de las necesidades en su camino a convertirse en software.

5.4 DESARROLLO DE LA PROPUESTA

El presente marco de trabajo de desarrollo de software sigue los lineamientos del manifiesto ágil. Este enfoque da una importancia central a la consideración de las personas involucradas en el proceso, reconociendo que son el recurso más valioso. Asimismo, se otorga una gran relevancia al software funcionando, comprendiendo que el software debe ser plenamente funcional y satisfacer las necesidades del cliente en todas las etapas del proyecto. La estrecha colaboración con el cliente es uno de los pilares fundamentales del desarrollo ágil; se fomenta una comunicación fluida y continua, lo que permite una comprensión profunda de las necesidades del cliente y la capacidad de adaptarse a cualquier cambio en los requisitos del proyecto; la colaboración constante se traduce en una mayor satisfacción del cliente y en la entrega de soluciones que realmente se ajustan a sus expectativas. Además, se mantiene un respeto inquebrantable por las libertades asociadas al Software Libre; esta adhesión a los principios del Software Libre no solo garantiza la transparencia y la ética en el trabajo realizado, sino que también promueve la innovación

y la colaboración en el ecosistema de Software Libre.

El Software Libre es algo más que un mero proceso técnico; desde la perspectiva de la ingeniería de software, se manifiesta como un proceso social en el que participan numerosas personas y organizaciones, independientemente de sus habilidades técnicas. En este contexto, se establece un ecosistema colaborativo en el que diversos actores, tanto técnicos como no técnicos, desempeñan un papel fundamental.

Desde un punto de vista técnico, la ingeniería de software desempeña un rol esencial en la creación y desarrollo de soluciones de Software Libre. Los expertos en este campo trabajan en estrecha colaboración con la comunidad de desarrolladores, aportando su conocimiento y habilidades para garantizar la calidad y el funcionamiento óptimo de las aplicaciones.

Sin embargo, es crucial destacar que el Software Libre va más allá de los aspectos técnicos. Se convierte en un proceso social inclusivo que atrae a individuos y organizaciones de diversas áreas y niveles de experiencia. Incluso aquellos sin habilidades técnicas pueden contribuir de manera significativa a través de actividades como la documentación, la promoción y el apoyo a la comunidad.

5.4.1. Ámbito de aplicabilidad

En el ámbito del desarrollo de software, es fundamental adoptar una perspectiva a largo plazo, no solo para abordar las necesidades presentes, sino también para anticipar las modificaciones y el mantenimiento futuros. La aplicación de un proceso de desarrollo de software adecuado no solo mejora la eficacia del equipo, sino que también aumenta significativamente las posibilidades de lograr un producto de software de alta calidad y duradero.

En el caso de equipos pequeños, la organización y la distribución de tareas pueden plantear desafíos adicionales. Sin embargo, es importante destacar que el presente proceso puede ser implementado con éxito en el desarrollo de Software Libre, por equipos reducidos. Esto se debe a que el enfoque y las prácticas establecidas en el proceso facilitan la coordinación y la colaboración efectiva, lo que resulta en un desarrollo de software más

fluido y de calidad.

5.4.2. Comunidad

La comunidad desempeña un papel de suma importancia en la dinámica de un proyecto de Software Libre. Antes de embarcarse en las actividades de ingeniería de software, resulta esencial comprender y participar activamente en la comunidad que respaldará y fortalecerá el producto de software.

En el contexto del desarrollo de Software Libre, la comunidad representa una red interconectada de individuos, colaboradores y usuarios que comparten un interés común en el software en cuestión. Esta comunidad no solo proporciona apoyo técnico y recursos valiosos, sino que también fomenta un ambiente de colaboración y aprendizaje mutuo.

Antes de dar inicio a cualquier proyecto de desarrollo de Software Libre, es imperativo que los involucrados se integren a esta comunidad de manera efectiva. Esto implica no solo familiarizarse con sus normas y prácticas, sino también contribuir activamente a sus discusiones, compartir conocimientos y establecer relaciones con otros miembros. Al hacerlo, se establece una base sólida para la colaboración y se aprovecha la experiencia colectiva de la comunidad para impulsar el éxito del proyecto.

Es necesario tener claro los siguientes aspectos:

Comunicación. En el contexto del proyecto de Software Libre, es fundamental comprender la diversidad de canales de comunicación disponibles, tanto asíncronos como síncronos, con el objetivo de facilitar una interacción efectiva con la comunidad. Entre los canales asíncronos se encuentran recursos como la página web del proyecto, listas de correo, foros y wikis, que proporcionan medios para compartir información, documentación y discutir temas de manera diferida, permitiendo la participación de miembros de la comunidad en diferentes zonas horarias y ritmos de trabajo.

Además de estos canales asíncronos, se deben considerar también las opciones síncronas, como la creación de grupos en plataformas de mensajería instantánea

como Telegram o Discord. Estos canales permiten una comunicación en tiempo real, lo que puede ser especialmente valioso para discusiones activas, resolución de problemas inmediatos y la creación de un sentido de comunidad más cercano y cohesionado.

Por otro lado, es esencial comprender y adherirse a las reglas de conducta y protocolo establecidas por la comunidad. Este conocimiento garantiza una comunicación efectiva y respetuosa entre los miembros del proyecto, promoviendo un ambiente colaborativo y armonioso. Respetar estas normas no solo contribuye a una comunicación efectiva, sino que también fortalece la relación de confianza dentro de la comunidad de Software Libre.

Técnico. En el ámbito del desarrollo de software dentro de la comunidad de Software Libre, es esencial adquirir un profundo conocimiento de todos los aspectos técnicos que rigen el proceso. Esto abarca una serie de áreas cruciales que incluyen, entre otras:

- Propuesta de nuevas características: Para contribuir de manera efectiva, es necesario comprender cómo se proponen nuevas características o mejoras en el software. Esto implica conocer los canales y procedimientos establecidos para presentar propuestas y discutirlas con la comunidad.
- Aprobación de propuestas: Comprender quiénes son los responsables de aprobar las propuestas y conocer el proceso de toma de decisiones es esencial para llevar a cabo contribuciones efectivas.
- Realización de aportes: Familiarizarse con el proceso de realizar aportes al código fuente es clave. Esto incluye aprender a utilizar sistemas de control de versiones como Git, entender los estándares de codificación y seguir prácticas de desarrollo colaborativo.
- Selección tecnológica: Conocer los lenguajes de programación, herramientas y tecnologías que se utilizan en el proyecto es esencial para desarrollar contribuciones efectivas y alineadas con la visión del software.
- Estilo de codificación: Adaptarse al estilo de codificación establecido en el

proyecto es necesario para mantener la coherencia y la legibilidad del código fuente.

- Acceso al código fuente: Saber cómo y dónde acceder al código fuente del proyecto es fundamental para comenzar a trabajar en él.
- Construcción y ejecución de versiones de desarrollo: Aprender cómo construir y ejecutar versiones de desarrollo del software es esencial para probar y validar las contribuciones.
- Reporte de incidencias: Entender cómo reportar problemas o incidencias de manera efectiva es importante para ayudar en la detección y solución de errores.
- Documentación del trabajo: La documentación adecuada de las contribuciones realizadas es crucial para que otros miembros de la comunidad comprendan el trabajo realizado y puedan colaborar de manera eficiente.

Soporte. Para desempeñar un papel eficaz en el desarrollo de Software Libre, es esencial comprender aspectos clave relacionados con el soporte del producto. Esto implica una serie de consideraciones importantes, que incluyen:

- Frecuencia de lanzamiento de versiones: Es crucial asimilar la frecuencia con la que se lanzan nuevas versiones del software. Esto proporciona una visión de la velocidad de desarrollo y permite a los colaboradores anticipar los cambios y las actualizaciones que pueden afectar su trabajo.
- Tiempo de mantenimiento de cada versión: Conocer el período de mantenimiento de cada versión del software es fundamental. Esto permite entender cuánto tiempo se proporcionará soporte y correcciones de seguridad para una versión específica antes de que se considere obsoleta.
- Cantidad de versiones disponibles al mismo tiempo: Comprender cuántas versiones del software están en uso simultáneamente en la comunidad es esencial. Esto puede variar según las necesidades y preferencias de los usuarios, y esta información es relevante para los desarrolladores y el equipo de soporte.

Este conocimiento es fundamental para garantizar que los colaboradores estén al tanto de los ciclos de desarrollo y mantenimiento del software, lo que les permite planificar y tomar decisiones informadas en cuanto a la contribución y el soporte continuo. Además, esta comprensión contribuye a una gestión más eficiente de los recursos y a la satisfacción general de los usuarios y miembros de la comunidad.

Legal. : En el ámbito del desarrollo de Software Libre, es imperativo comprender los aspectos legales que rigen el proyecto. Esto abarca una serie de consideraciones legales esenciales, que incluyen:

- Licencia utilizada por el proyecto: Es fundamental tener un conocimiento profundo de la licencia bajo la cual se distribuye el software. Esta licencia establece los términos y condiciones para el uso, distribución y modificación del software, y es esencial para garantizar el cumplimiento legal.
- Derechos de autor: Comprender los derechos de autor asociados al software es crucial. Esto incluye la identificación de los autores y propietarios de derechos, así como el reconocimiento de las contribuciones individuales y colectivas al proyecto.
- Legislación aplicable: Es necesario estar al tanto de la legislación vigente que puede resolver posibles conflictos legales relacionados con el software. Esto implica conocer las leyes de derechos de autor, las regulaciones de propiedad intelectual y otras normativas relevantes en la jurisdicción pertinente.
- Conocimiento de las licencias: Todos los miembros del equipo deben familiarizarse con las licencias utilizadas en el proyecto y comprender las libertades y restricciones que implican. Esto es esencial para garantizar un uso y distribución legales del software.
- Selección de componentes: Se debe tener precaución al seleccionar componentes o bibliotecas de software para su inclusión en el proyecto, ya que existen licencias que pueden ser incompatibles con la licencia del software principal. La elección cuidadosa de componentes es fundamental para evitar conflictos legales.

El cumplimiento de estos aspectos legales es esencial para mantener la integridad del proyecto de Software Libre y evitar posibles litigios o problemas legales. Además, garantiza que el software esté disponible y utilizable en conformidad con los principios del Software Libre, lo que es esencial para la comunidad de usuarios y desarrolladores.

5.4.3. Actividades

Las actividades se dividen en tres momentos estrechamente interconectados, los cuales recaen en la responsabilidad de todos los miembros del equipo, quienes deben llevar a cabo estas tareas de la manera más eficiente y efectiva posible. En equipos de menor tamaño, la carga de trabajo incluye la ejecución de todas las etapas del ciclo de desarrollo del proyecto, que abarcan desde la captura inicial de los requerimientos hasta la fase de puesta en producción.

Estos tres momentos, que se interrelacionan de forma sinérgica, conforman un enfoque integral para el desarrollo de software. Comprender la naturaleza interdependiente de estas etapas es crucial para el éxito del proyecto. Cada miembro del equipo debe asumir su responsabilidad en la ejecución de estas actividades para garantizar que el proceso se lleve a cabo de manera fluida y eficiente.

Implementación

El objetivo central de la etapa de implementación consiste en desarrollar el software con el más alto estándar de calidad, garantizando que cumpla plenamente con los requisitos establecidos por los usuarios.

Para transformar el requerimiento, que se define como una historia de usuario detallada, en un software funcional y beneficioso, resulta esencial cumplir con todos los criterios de aceptación y sus respectivos escenarios. Estos criterios de aceptación son las condiciones y expectativas específicas que el software debe satisfacer para ser considerado exitoso en términos de satisfacer las necesidades y expectativas de los usuarios.

La implementación no solo se enfoca en traducir el requerimiento en código, sino que también se esfuerza en asegurar que el software resultante se ajuste con precisión a las especificaciones y requisitos definidos. Esto implica una cuidadosa atención a los detalles y una validación constante a medida que se desarrolla el software, con el objetivo de asegurar que se cumplan todos los criterios de aceptación y que los escenarios definidos se manejen de manera adecuada.

Para satisfacer con las necesidades de un escenario se siguen tres pasos, como se puede ver la figura V.1:

Figura V.1: Proceso de Implementación.
Elaboración propia.

Escribir prueba unitaria. El primer paso es la creación de una prueba unitaria que tiene como objetivo validar el cumplimiento de un escenario específico. Esta prueba debe ser diseñada de manera simple y directa, de modo que inicialmente falle cuando se ejecute. La redacción de la prueba antes que la implementación del código en sí aporta diversos beneficios, siendo uno de los más destacados el mayor entendimiento del requerimiento.

Definir primero la prueba unitaria proporciona una oportunidad para una comprensión más profunda del requisito en cuestión. En este enfoque, el desarrollador se concentra en la validación de lo que el usuario verdaderamente necesita en lugar de probar simplemente lo que el software es capaz de hacer. Esto permite un alineamiento más estrecho con las expectativas del usuario y contribuye a la creación de software que cumple de manera efectiva con las necesidades del usuario final.

La prueba unitaria establece una base sólida para el proceso de desarrollo al enfo-

carse en la validación temprana y continua, lo que reduce la posibilidad de errores y garantiza un software de mayor calidad y precisión.

Escribir código. El siguiente paso consiste en redactar el código fuente necesario para que la prueba unitaria previamente definida pueda ejecutarse de manera satisfactoria. En este punto, el objetivo principal es crear la versión más sencilla del código que garantice que la prueba pase de manera correcta.

La simplicidad en esta etapa es clave, ya que se busca desarrollar una solución eficaz y directa para abordar el escenario específico definido en la prueba unitaria. En lugar de buscar complejidades innecesarias, se enfoca en proporcionar una implementación que satisfaga los requisitos mínimos necesarios para que la prueba sea exitosa.

Este enfoque de escribir el código de manera simple contribuye a la claridad y la mantenibilidad del software, ya que reduce la posibilidad de errores y facilita la comprensión tanto para el desarrollador como para otros miembros del equipo. Además, al seguir este proceso paso a paso, se asegura que cada componente del software esté alineado con las necesidades reales del usuario y se evita la sobrecarga innecesaria.

Refactorización. Finalmente, el último paso es la refactorización, que se enfoca en mejorar el código fuente existente, asegurando al mismo tiempo que continúe pasando las pruebas definidas previamente. La refactorización es una práctica esencial para mantener y mejorar la calidad del código a lo largo del ciclo de vida del proyecto.

Durante la refactorización, el equipo de desarrollo revisa el código existente en busca de oportunidades de mejora en términos de eficiencia, claridad y mantenibilidad. Esto puede incluir la eliminación de duplicaciones, la simplificación de estructuras complejas, la mejora de nombres de variables y funciones, y la aplicación de buenas prácticas de codificación.

Es importante destacar que, a pesar de realizar cambios en el código, la refactorización no debe comprometer la funcionalidad existente. Es decir, el código debe seguir pasando las pruebas de manera exitosa, lo que garantiza que las modificacio-

nes no introduzcan errores ni afecten negativamente la integridad del software.

La refactorización es una práctica clave en el desarrollo de software, ya que contribuye a mantener un código limpio, comprensible y eficiente a lo largo del tiempo. Además, ayuda a prevenir la acumulación de deudas técnicas, lo que puede dificultar el desarrollo futuro y la evolución del software.

Una vez que la fase de refactorización ha finalizado con éxito, el proceso continúa mediante la repetición de las etapas anteriores, agregando una nueva prueba unitaria en cada iteración. Este enfoque iterativo es fundamental para garantizar la robustez y la calidad del software desarrollado.

La adición de una nueva prueba unitaria en cada ciclo permite abordar progresivamente todos los escenarios y requisitos del software. Se busca identificar y resolver problemas de manera incremental, lo que facilita la detección temprana de posibles errores o incompatibilidades a medida que se añaden nuevas funcionalidades.

Cuando se llega al punto en el que ya no es posible agregar nuevas pruebas unitarias para un escenario específico y el código existente satisface todas las pruebas exitosamente, se considera que ese escenario ha sido implementado de manera satisfactoria. A partir de este punto, se puede avanzar al siguiente escenario o requisito pendiente.

Este enfoque de desarrollo iterativo y basado en pruebas garantiza que cada parte del software se someta a una validación exhaustiva y que todas las funcionalidades se implementen con la máxima calidad posible. Además, al abordar los requisitos de manera gradual, se reduce la complejidad y se facilita la identificación y corrección de problemas en las primeras etapas del proceso de desarrollo.

La etapa de implementación llega a su conclusión una vez que la historia de usuario ha sido programada y sometida a pruebas exitosas. Esto significa que todos los criterios de aceptación y escenarios definidos cuentan con pruebas que los validan y que el código fuente correspondiente ha superado satisfactoriamente estas pruebas. Este enfoque garantiza que el software desarrollado alcance un alto nivel de calidad y funcione de manera confiable.

Una vez que la implementación cumple con estos criterios, el código se somete a un

proceso de revisión por parte de otros miembros del equipo. Esta revisión busca garantizar que el código esté escrito de manera clara, siga las mejores prácticas de codificación y cumpla con los estándares establecidos por el proyecto. Una vez que el código ha sido revisado y aprobado, se procede a su incorporación en la rama principal del repositorio de código del proyecto.

Esta fase de revisión y unión del código principal es esencial para mantener la integridad y la cohesión del proyecto de desarrollo de software. A través de este proceso, se integran las contribuciones individuales en la rama principal, lo que permite que el software evolucione de manera continua y que todas las mejoras y características sean parte del producto final.

Se aconseja que el desarrollador sea un programador con experiencia en TDD.

Especificación

El propósito central de la etapa de especificación es guiar el desarrollo del software hacia la consecución de la más alta calidad posible, asegurando que cumpla con el comportamiento necesario y esperado por parte de los usuarios.

En esta etapa, se define de manera precisa y detallada el comportamiento requerido por los usuarios. Esto implica establecer de manera clara los requisitos funcionales y no funcionales del software, así como cualquier otra especificación relevante. La especificación actúa como un conjunto de directrices que orienta el proceso de desarrollo hacia la creación de un producto que se ajuste a las necesidades y expectativas de los usuarios finales.

La calidad en esta etapa implica la claridad y la exhaustividad de la especificación. Cuanto más precisa y completa sea la especificación, más sólida será la base para el desarrollo de software de alta calidad. Esto ayuda a prevenir malentendidos y asegura que el software cumpla con los requisitos establecidos desde el principio.

Cada requerimiento, definido como una historia de usuario, para convertirse en software útil debe seguir tres pasos, como se puede ver la figura V.2

Especificar requerimiento. La especificación de requerimientos implica un proceso de

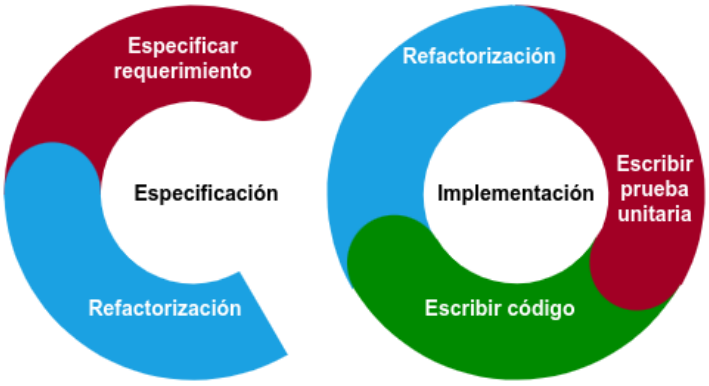

Figura V.2: Proceso de Especificación.
Elaboración propia.

enriquecimiento de cada criterio de aceptación de la historia de usuario. Esto se logra mediante la inclusión de escenarios, pasos y ejemplos específicos, que se definen como pruebas de aceptación. Estas pruebas desempeñan un papel crucial al permitir una verificación efectiva del comportamiento del software, garantizando que cumpla con las expectativas de los usuarios.

Para automatizar esta verificación, se recomienda el uso de un lenguaje específico de dominio, como Gherkin, junto con herramientas que faciliten la ejecución de pruebas de manera rápida y eficiente. La automatización de pruebas no solo agiliza el proceso de verificación, sino que también contribuye a la detección temprana de posibles problemas.

Además de las pruebas de aceptación, es necesario definir pruebas de componentes, pruebas de integración y pruebas de interfaz de programación de aplicaciones (API) para garantizar un alto nivel de calidad y confiabilidad en todas las capas del software. Es importante destacar que si bien las pruebas de interfaz de usuario son esenciales, su ejecución puede ser costosa en términos de tiempo y recursos, por lo que se recomienda escribir solo las pruebas estrictamente necesarias para cubrir los aspectos críticos de la interfaz de usuario.

Implementación. Aplicar la implementación a la historia de usuario especificada.

Refactorización. Un aspecto fundamental en el proceso de desarrollo es la mejora continua de las pruebas de aceptación, que implica la identificación de pasos que se

repiten en diferentes pruebas. Esta práctica busca optimizar y simplificar las pruebas, promoviendo la eficiencia y la mantenibilidad del conjunto de pruebas.

Durante este proceso de mejora, se revisan detenidamente las pruebas de aceptación existentes para identificar cualquier redundancia o repetición de pasos entre ellas. Estos pasos comunes pueden incluir acciones que se ejecutan de manera similar en varias pruebas, lo que conduce a una duplicación innecesaria de esfuerzo y recursos.

Una vez identificados estos pasos repetitivos, se toma la decisión de refactorizar las pruebas, lo que implica la consolidación de los pasos comunes en una sola prueba o la creación de funciones o módulos reutilizables que puedan ser invocados desde múltiples pruebas. Esta consolidación y reutilización contribuye a simplificar y optimizar el conjunto de pruebas, reduciendo la complejidad y mejorando la mantenibilidad.

La mejora de las pruebas de aceptación no solo aumenta la eficiencia del proceso de prueba, sino que también facilita la detección de problemas y la corrección de errores, ya que los cambios necesarios solo deben realizarse en un lugar centralizado. Esto garantiza que las pruebas sigan siendo efectivas y confiables a medida que el software evoluciona.

La etapa de especificación llega a su término una vez que se han verificado con éxito todos los escenarios definidos y el software se ajusta completamente al comportamiento esperado. Esto representa un hito fundamental en el proceso de desarrollo, ya que indica que el software ha sido desarrollado en conformidad con los requisitos y las expectativas establecidas por los usuarios.

Para lograr esta finalización, se lleva a cabo un proceso exhaustivo de verificación que abarca cada uno de los escenarios y casos de prueba definidos previamente. Cada escenario se ejecuta y se valida para asegurarse de que el software se comporte de acuerdo con lo requerido y esperado.

Al alcanzar este punto, se obtiene la certeza de que el software cumple con los estándares de calidad y funcionalidad establecidos en la especificación. Esto es esencial para garantizar que el producto final sea capaz de satisfacer las necesidades y demandas de los

usuarios de manera efectiva y confiable.

Un componente esencial del proceso implica la evaluación y revisión del trabajo realizado por el equipo, con el fin de identificar tanto los aspectos positivos como los negativos. Esta evaluación tiene como objetivo principal maximizar los logros y minimizar las deficiencias a través de acciones específicas.

En esta revisión, se lleva a cabo un análisis exhaustivo de las contribuciones individuales y colectivas del equipo. Se identifican y se destacan los aspectos positivos, como los logros sobresalientes, la eficiencia en la ejecución de tareas y la colaboración efectiva entre los miembros. Al mismo tiempo, se prestan especial atención a los aspectos negativos, como posibles retrasos, errores o áreas que requieren mejoras.

Una vez identificados estos aspectos, se desarrollan estrategias y acciones concretas para capitalizar los aspectos positivos y mitigar los aspectos negativos. Esto puede incluir la implementación de prácticas exitosas, la resolución de problemas específicos o la asignación de recursos adicionales cuando sea necesario.

Esta revisión periódica y reflexiva del trabajo del equipo contribuye al crecimiento y al mejoramiento continuo del proceso de desarrollo. Proporciona una oportunidad para el aprendizaje y la adaptación, lo que resulta en un equipo más eficiente y en la obtención de resultados de mayor calidad.

Se aconseja que el desarrollador sea un probador con experiencia en BDD.

Exploración

El propósito fundamental de la etapa de exploración es proporcionar a la comunidad cambios y mejoras en el software que mantengan un alto estándar de calidad y que incluyan funcionalidades capaces de satisfacer las necesidades y expectativas de los miembros. Esta etapa representa un proceso continuo de evolución y transformación del software, que asegura que tanto el proyecto como el software se mantengan relevantes y vigentes.

La transformación del software se lleva a cabo a través de varias acciones, que pueden incluir la incorporación de nuevas características que amplíen la funcionalidad del software,

la modificación de características existentes para mejorar su rendimiento o adaptarlas a nuevas necesidades, así como la corrección de errores conocidos que puedan afectar la experiencia del usuario.

La exploración se basa en la retroalimentación y las contribuciones de la comunidad de usuarios y desarrolladores, que desempeñan un papel activo en la identificación de áreas de mejora y en la sugerencia de nuevas características. Esta colaboración continua entre la comunidad y el equipo de desarrollo garantiza que el software evolucione de manera acorde con las demandas cambiantes y las expectativas de los usuarios.

En el seno de la comunidad, es común encontrar una diversidad de intereses y necesidades que a menudo no pueden ser atendidos de manera satisfactoria debido a que un número reducido de miembros efectivamente contribuyen con trabajo técnico, particularmente en el ámbito de la ingeniería de software. En consecuencia, se torna esencial identificar y abordar de manera sistemática las necesidades que deben ser convertidas en código funcional en futuras versiones del software.

Es precisa la identificación y documentación de las necesidades que surgen dentro de la comunidad. Estas necesidades pueden ser variadas y pueden surgir de diversas fuentes, como sugerencias de usuarios, informes de errores, o requerimientos específicos de grupos o individuos.

Una vez que estas necesidades han sido identificadas, se procede a su agrupación y priorización. Esto implica clasificar las necesidades en categorías según su tipo y luego asignarles un valor y un nivel de impacto en el proyecto. Las necesidades críticas o de alta prioridad se abordan antes que aquellas de menor prioridad, lo que asegura que los recursos se utilicen de manera efectiva y se satisfagan las necesidades más importantes de la comunidad.

Para mitigar una necesidad a través de software útil es necesario seguir tres pasos, como se puede ver en la figura V.3:

Definir requerimiento. La transformación de una necesidad en un elemento formal del software implica la creación de un requerimiento bien definido. Este proceso consiste en la elaboración de una historia de usuario que actúe como un modelo que

Figura V.3: Proceso de Exploración.
Elaboración propia.

represente el requerimiento de manera clara y precisa.

La historia de usuario es una herramienta fundamental en el proceso de desarrollo, ya que describe de manera detallada el contexto, los objetivos y las expectativas relacionadas con el requerimiento. En esta historia se incluyen elementos como los actores involucrados, los criterios de aceptación y cualquier información relevante que permita comprender a fondo el requerimiento.

La definición de un requerimiento de esta manera asegura que todas las partes interesadas, tanto dentro del equipo de desarrollo como en la comunidad de usuarios, tengan una comprensión común y compartida del objetivo que se busca alcanzar. Además, proporciona una base sólida para el diseño y la implementación del software.

Especificación. Aplicar la especificación a la historia de usuario.

Lanzamiento. El lanzamiento consiste en poner a disposición de la comunidad el software resultante, permitiendo que los usuarios lo utilicen y proporcionen retroalimentación valiosa. Este proceso es esencial para obtener una visión completa de cómo el software se desempeña en situaciones reales y para identificar posibles errores o áreas de mejora.

Durante esta etapa, el software se pone a disposición de los usuarios, ya sea a través de descargas, actualizaciones o implementaciones en línea, dependiendo de la naturaleza del proyecto. Se fomenta activamente la participación de la comunidad,

animándolos a utilizar el software y a compartir sus experiencias y comentarios.

La retroalimentación de la comunidad desempeña un papel fundamental en la mejora continua del software. Los usuarios pueden informar sobre cualquier error que encuentren, proporcionar sugerencias de funcionalidades adicionales o expresar sus necesidades y expectativas. Esta retroalimentación es valiosa para el equipo de desarrollo, ya que permite abordar problemas de manera proactiva y realizar ajustes o correcciones según sea necesario.

El lanzamiento también puede incluir la documentación actualizada, tutoriales y recursos de soporte para ayudar a los usuarios a sacar el máximo provecho del software. Esto contribuye a una experiencia de usuario positiva y efectiva.

Tras la finalización del lanzamiento, se procede a realizar un análisis exhaustivo de la retroalimentación proporcionada por la comunidad de usuarios. El objetivo principal de este análisis es mejorar la definición del requerimiento y trabajar en pos de alcanzar la versión más óptima posible de la funcionalidad, dentro del tiempo disponible para el desarrollo del incremento.

La retroalimentación recibida de la comunidad es un recurso invaluable para el equipo de desarrollo, ya que proporciona información real y concreta sobre cómo los usuarios interactúan con el software en su entorno de uso real. Esta información puede incluir la detección de errores no identificados previamente, la identificación de necesidades no satisfechas y sugerencias para mejorar la funcionalidad existente.

El proceso de mejora continua comienza con la revisión detallada de esta retroalimentación. Se identifican los patrones y temas recurrentes, así como los comentarios específicos, para comprender a fondo los desafíos y las oportunidades de mejora. A partir de esta comprensión, se pueden realizar ajustes en la definición del requerimiento, incorporando nuevos elementos o modificando los existentes para abordar de manera más efectiva las necesidades de los usuarios.

Es importante destacar que este proceso de mejora es iterativo y puede involucrar múltiples ciclos de retroalimentación y ajustes. El objetivo final es alcanzar la mejor versión posible de la funcionalidad, aprovechando la información valiosa proporcionada por la

comunidad de usuarios.

La etapa de exploración llega a su conclusión cuando todas las necesidades identificadas han sido transformadas en software funcional y se han empaquetado para su liberación. En este punto, el software se presenta como una versión completa y lista para ser utilizada por cualquier persona, ya sea miembro de la comunidad o no.

Durante la exploración, se han abordado y satisfecho todas las necesidades identificadas, lo que implica la implementación de nuevas características, la modificación de funcionalidades existentes y la corrección de errores conocidos. Estos cambios y mejoras se han desarrollado en respuesta a la retroalimentación de la comunidad y a las necesidades emergentes.

El proceso culmina con la preparación y empaquetado del software en una forma que sea fácilmente accesible y utilizable por los usuarios. Esto puede implicar la creación de instaladores, la generación de documentación actualizada y la publicación de la versión en un repositorio o plataforma adecuada.

El software resultante se presenta como una solución lista para ser aprovechada por cualquier persona o institución, sin restricciones en cuanto a quién puede acceder a él. Esto refleja el compromiso de la exploración de ofrecer un producto útil y de alta calidad que esté al alcance de todos.

5.4.4. Artefactos

Historias de usuario

Las historias de usuario son un componente fundamental para capturar y definir los requerimientos de los usuarios. Cada historia de usuario debe incluir los siguientes elementos:

Título. El título de la historia debe ser breve y descriptivo, proporcionando una idea clara de la funcionalidad que se solicita. Este título actúa como una etiqueta identificativa para la historia.

Descripción. La descripción de la historia de usuario es el componente que proporciona

un detalle completo de la funcionalidad requerida. Debe estar redactada utilizando un lenguaje claro y comprensible para que todos los miembros del equipo puedan entender la naturaleza y el alcance de la funcionalidad.

Criterios de aceptación. Los criterios de aceptación son elementos cruciales que permiten determinar de manera objetiva si la funcionalidad ha sido completada con éxito. Cada criterio debe ser específico y mensurable, y se apoya en ejemplos y escenarios que facilitan la comprensión y verificación del cumplimiento de la funcionalidad.

Prioridad. La prioridad asignada a una historia de usuario refleja el grado de valor que aporta a los usuarios. En general, las historias con una prioridad más alta se implementan primero, lo que permite abordar las necesidades más importantes o urgentes antes que otras.

Estimación. La estimación representa el esfuerzo necesario por parte del desarrollador para convertir la historia de usuario en software funcional y útil. Esta estimación ayuda al equipo a planificar y asignar recursos de manera efectiva durante el desarrollo.

Código fuente

En el contexto de un proyecto de Software Libre, es común observar que este no comienza desde cero, sino que se basa en el código fuente y el trabajo previo de otros proyectos. Esta estrategia se adopta con el propósito de evitar la duplicación de esfuerzos y optimizar la colaboración y el aprovechamiento de los recursos dentro de la comunidad.

En lugar de comenzar desde cero, los proyectos de Software Libre se benefician al aprovechar el conocimiento, las soluciones técnicas y el código fuente desarrollado por otros miembros de la comunidad o proyectos relacionados. Esto permite acelerar el proceso de desarrollo, ya que se construye sobre una base existente y probada, en lugar de reinventar la rueda.

La reutilización de código y la colaboración entre proyectos son prácticas fundamentales en la filosofía del Software Libre. Los desarrolladores pueden integrar componentes de código abierto, bibliotecas, módulos y soluciones existentes en su propio proyecto, siem-

pre respetando las licencias y los términos de uso. Esto no solo ahorra tiempo y recursos, sino que también contribuye al enriquecimiento continuo del ecosistema de Software Libre.

El código fuente, indiscutiblemente, ocupa un lugar central y fundamental en el contexto del desarrollo de Software Libre, ya que proporciona acceso directo a dos de las libertades. El código fuente desempeña un papel crucial en la capacidad de definir las instrucciones que serán generadas por el software, el cual, a su vez, será ejecutado por el hardware con el propósito de satisfacer las necesidades de los usuarios.

En primer lugar, el código fuente es el medio a través del cual se expresan las instrucciones y algoritmos que guían el comportamiento del software. Esto significa que, al tener acceso al código fuente, los desarrolladores pueden comprender y modificar el funcionamiento interno del software de manera transparente y personalizada. Esto brinda a los usuarios un nivel de control sin precedentes sobre las soluciones de software, lo que es esencial para satisfacer sus necesidades específicas.

Además, el código fuente desempeña un papel crítico en el aseguramiento de la calidad del software. Al representar las pruebas automatizadas que verifican el cumplimiento de todas las necesidades y comportamientos de los usuarios, el código fuente contribuye a garantizar que el software sea confiable, libre de errores y cumpla con los estándares de calidad establecidos. Estas pruebas ayudan a mantener un alto nivel de confiabilidad y robustez en el software, lo que es esencial para la satisfacción de los usuarios.

El código fuente desempeña un papel esencial al proporcionar una representación objetiva y tangible del estado de avance de un proyecto de desarrollo de software. En comparación con una documentación extensa, el software en funcionamiento, incluso si presenta algunos defectos, permite alcanzar los objetivos de manera más rápida y efectiva.

El código fuente actúa como una representación concreta de las implementaciones realizadas en el proyecto. A medida que se desarrolla y se agregan nuevas funcionalidades, el código fuente evoluciona de manera directa y refleja el progreso real del proyecto. Esta transparencia y tangibilidad son invaluables tanto para los miembros del equipo de desarrollo como para la comunidad de usuarios.

Cuando se presenta software funcional, incluso si aún contiene algunos defectos, los usuarios pueden interactuar con él de inmediato y evaluar su utilidad y capacidad para satisfacer sus necesidades. Esto permite obtener retroalimentación valiosa y orientación sobre qué áreas requieren atención adicional y mejora. Además, proporciona una forma concreta de medir el progreso del proyecto en función de la funcionalidad implementada.

En contraste, una documentación extensa puede ser útil, pero a menudo se experimenta como una representación abstracta del proyecto. Puede requerir tiempo adicional para traducir la documentación en una implementación real y funcional. El software en funcionamiento, incluso con defectos, agiliza la evaluación y la retroalimentación, lo que ayuda al equipo de desarrollo a responder de manera más ágil a los cambios y a las necesidades de los usuarios.

Tablero Kanban

Es una herramienta que facilita la visualización clara y efectiva del estado y la prioridad de una necesidad a medida que progresa a través de las diversas etapas que conforman el proceso de transformación, hasta su culminación como software funcional y útil.

El tablero Kanban se convierte en una representación visual dinámica del flujo de trabajo del equipo de desarrollo. Cada necesidad se representa como una tarjeta en el tablero, y el equipo la mueve de una columna a otra a medida que avanza en su desarrollo. Estas columnas suelen representar las tareas por hacer, las que están en proceso y las realizadas, como se puede ver en la figura V.4.

Esta herramienta ofrece múltiples ventajas. En primer lugar, permite una visualización clara y en tiempo real del estado de cada necesidad. Los miembros del equipo y los stakeholders pueden identificar rápidamente en qué etapa se encuentra una necesidad y cuál es su prioridad en relación con otras. Además, promueve la transparencia y la colaboración al brindar una vista compartida del progreso del proyecto.

El uso del tablero Kanban también facilita la gestión eficiente del flujo de trabajo, ya que ayuda a evitar cuellos de botella y a asignar recursos de manera efectiva. Además, promueve una comunicación fluida entre los miembros del equipo, lo que facilita la toma

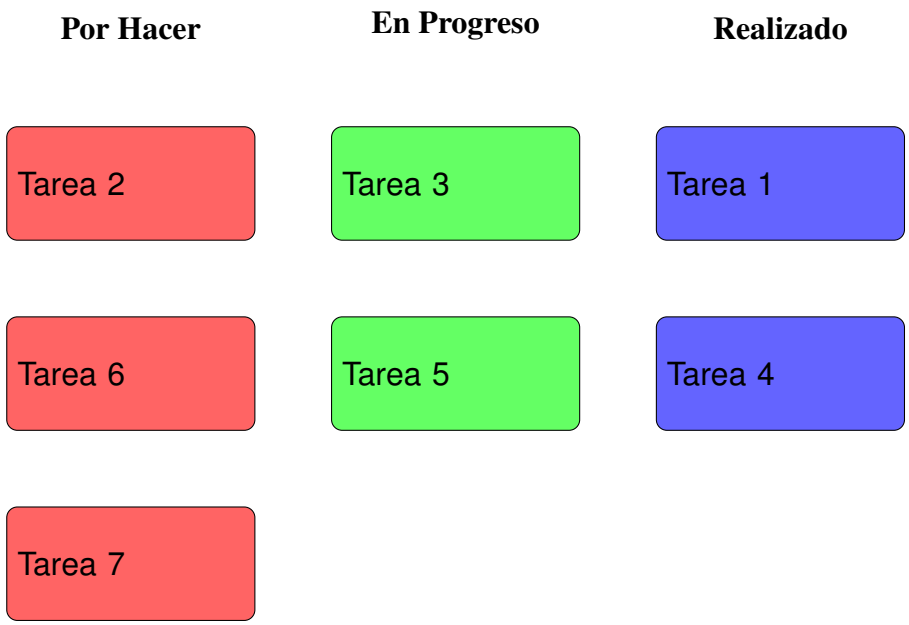

Figura V.4: Tablero Kanban
Fuente: Elaboración Propia.

de decisiones informadas y la resolución de problemas de manera ágil.

ANEXO A

EXPERTOS

Experto	Experiencia profesional	Titulo Académico	Grado Académico	Lugar de trabajo	Cargo
1	9	Ing. De Sistemas	Magister en Educación Superior	Universidad San Francisco Xavier	Docente
2	10	Ing. De Sistemas	Licenciado	Salesforce	Developer
3	8	Ing. De Sistemas	Magister en Tecnología de la Información y Seguridad	DAF - Órgano Judicial	Profesional en Administración de Sistemas y Comunicaciones
4	18	Ing. De Sistemas	Licenciado	IT Group Systems	CTO
5	12	Ing. De Sistemas	Licenciado	Tribunal Supremo Electoral	Profesional 3 padrón electoral
6	12	Ing. De Sistemas	Licenciado	Gobierno Autónomo Municipal de Sucre	Desarrollador de software

BIBLIOGRAFÍA

Agile Alliance (2022). Acceptance test driven development (atdd).

Audacity (2021). The feature requests pipeline.

Barrientos, A. (2022). Plan de versiones programadas.

Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., and Thomas, D. (2001a). Manifesto for agile software development.

Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., and Thomas, D. (2001b). Principles behind the agile manifesto.

Blé, C. (2010). Diseño Ágil con tdd.

Bolivia (2011). Ley nº 164. ley general de telecomunicaciones, tecnologías de información y comunicación. *Gaceta Oficial del Estado Plurinacional de Bolivia*.

Garzas, J. (2015). ¿qué es eso de atdd?

Haddad, I. (2008). Open source development model.

Haddad, I. and Warner, B. (2011). Understanding the open source development model.

Iglesias, F. (2021). Aprende test driven development.

Jääskeläinen, F. G. (2013). Methodologies used in open source approach to developing software in companies in finland.

Machuca-Villegas, L., Gasca-Hurtado, G. P., Puente, S. M., and Tamayo, L. M. R. (2021). Factores sociales y humanos que influyen en la productividad del desarrollo de soft-

ware: Medición de la percepción. *Revista Ibérica de Sistemas e Tecnologias de Informação*, (E41):488–502.

Newman, A. (2017). An inside look at brave development.

Parker, J. (2022). Best open source software of 2022.

Pressman, R. S. (2010). Ingeniería de software enfoque practico. pressman. pdf. *Ingeniería del software, un enfoque práctico*.

Ramírez, I. (2013). Apuntes de metodología de la investigación: un enfoque crítico. *Sucre, Bolivia*, 4.

Ramos, C. A. (2015). Los paradigmas de la investigación científica. *Avances en psicología*, 23(1):9–17.

Saldarriaga-Zambrano, P. J., Bravo-Cedeño, G. d. R., and Loor-Rivadeneira, M. R. (2016). La teoría constructivista de jean piaget y su significación para la pedagogía contemporónea. *Domino de las Ciencias*, 2(3 Especial):127–137.

SmartBear Software (2022). Behaviour-driven development.

Terhorst-North, D. (2006). Introducing bdd.

The kernel development community (2022). How the development process works¶.

Toledo, F. (2017). ¿qué es bdd?

Uribe, E. H. and Ayala, L. E. V. (2007). Del manifiesto ágil sus valores y principios. *Scientia et technica*, 13(34):381–386.