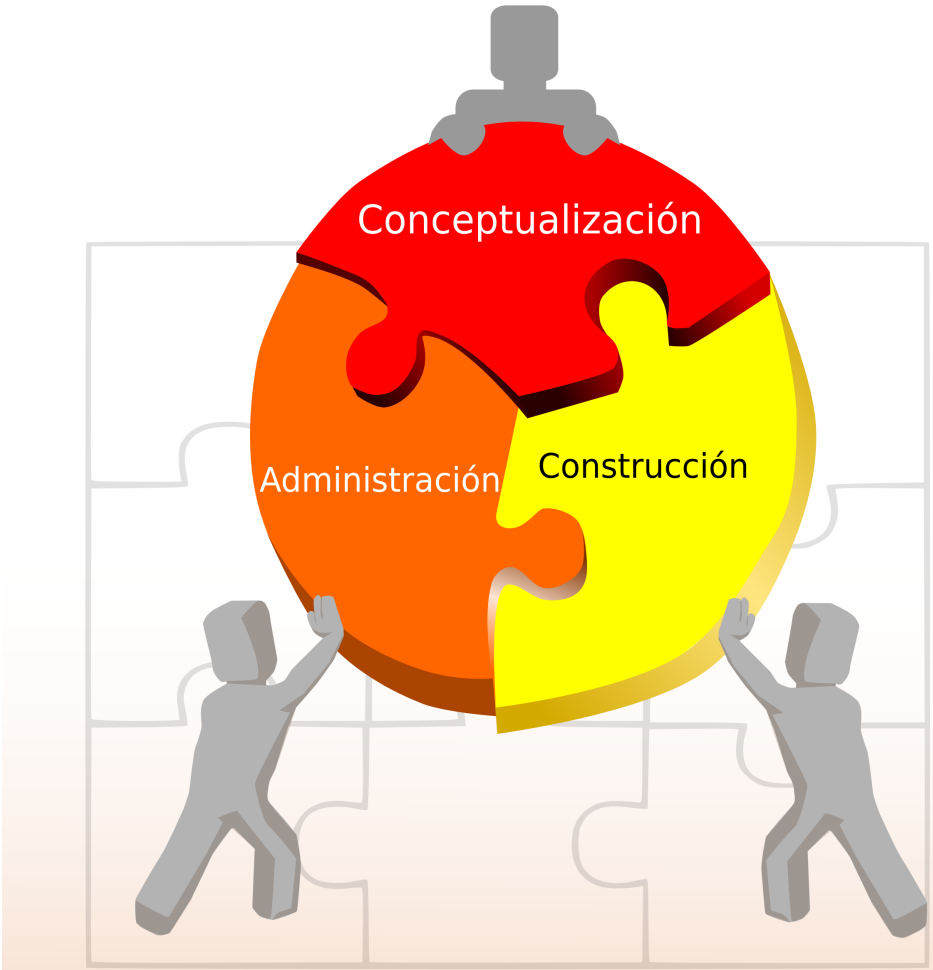
CENTRO NACIONAL DE DESARROLLO
E INVESTIGACIÓN EN TECNOLOGÍAS LIBRES (CENDITEL)

CENDITEL

Metodología
para el Desarrollo Colaborativo de Software Libre



Gobierno Bolivariano de Venezuela | Ministerio del Poder Popular para la Educación Universitaria, Ciencia y Tecnología | Centro Nacional de Desarrollo e Investigación en Tecnologías Libres (Cenditel)

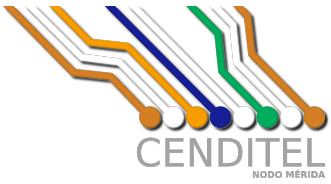
METODOLOGÍA PARA EL DESARROLLO COLABORATIVO DE SOFTWARE LIBRE

METODOLOGÍA PARA EL DESARROLLO COLABORATIVO DE SOFTWARE LIBRE

Versión 2

Johanna Álvarez, Víctor Bravo.
Fundación Centro Nacional de Desarrollo e Investigación en Tecnologías Libres

CENDITEL
Ministerio del Poder Popular para la Educación Universitaria, Ciencia y Tecnología



Publicación de la Fundación CENDITEL

Julio 2015, Primera Edición

ISBN de la obra independiente: No. 978-980-7154-19-2 Deposito Legal No. lfi7012015004595

Portada: Cipriano Alvarado

Maquetado usuando LATEX, BibTEX y pdfTEX.

A Manuela y Marcelo

COLABORADORES

MPPCTI Ministerio del Poder Popular para la Ciencia, Tecnología e Innovación

En la elaboración de la segunda versión de la Metodología para el Desarrollo Colaborativo de Software Libre se contó con el apoyo de miembros del Equipo de Desarrollo de la Fundación CENDITEL. Cabe destacar que el apoyo brindado se basó en la aplicación de la primera versión de la metodología en proyectos de desarrollo de esta Fundación, a partir de la cual surgieron importantes observaciones y recomendaciones que dieron paso a esta segunda versión. En ese sentido agradecemos a los compañeros:

- Solazver Solé.
- Antonio Araujo.
- Dhionel Díaz.
- Santiago Roca.
- Nelevis Baez.
- Rodolfo Sumoza.
- Cipriano Alvarado.

Índice general

LISTA DE FIGURAS

LISTA DE TABLAS

INTRODUCCIÓN

VÍCTOR BRAVO, ING.

Fundación Centro Nacional de Desarrollo e Investigación en Tecnologías Libres
Venezuela

En los últimos años la necesidad de contar con más y mejor software ha venido incrementándose de manera vertiginosa. La incorporación del cálculo computacional en cada vez más tareas ha sido impulsado por la aparición de las redes sociales, así como por otros fenómenos que han hecho que las tecnologías de la información invadan prácticamente todos los aspectos de nuestras vidas. Dado este panorama, la mejora continua en los procesos de creación de software es una labor ineludible. El desarrollo de software ha sido abordado desde diferentes ópticas y se ha adaptado a los ambientes de alta carga de trabajo, equipos multidisciplinarios y a tiempos cada vez más estrechos para las entregas.

Por otro lado, el software libre[1] se ha consolidado como forma de producción, hecho que introduce nuevos retos vinculados al área de la gestión de procesos de software.

En relación a lo antes expuesto, en este libro se presenta una segunda versión de la Metodología para el Desarrollo Colaborativo de Software Libre. Este nuevo documento es producto de una revisión amplia del primer material publicado en el año 2007, así como también de una evaluación práctica de los procesos planteados en la metodología, lo que ha dado como resultado la incorporación de nuevas técnicas de gestión, la simplificación de flujos de trabajo, la adición de actividades no contempladas inicialmente, tales como el empaquetado y la gestión de versiones, y una actualización de la lista de herramientas informáticas que incluye la disponibilidad de un complemento para la plataforma *Trac*[2] que está específicamente diseñado para la aplicación de esta metodología.

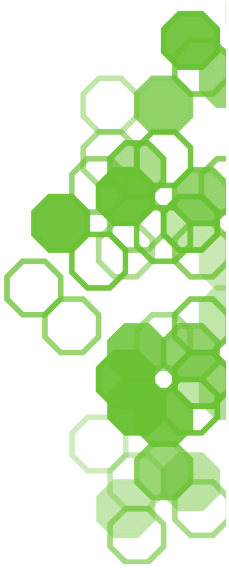
Los procesos que conforman la metodología están orientados a la conceptualización, administración y construcción (éste último anteriormente llamado desarrollo) de aplicaciones de software libre. En este trabajo se describen estos procesos de forma ordenada, pero tomando en cuenta su carácter iterativo, donde las activi-

[1] El software libre se define bajo cuatro libertades que se pueden consultar de forma amplia en el enlace http://www.gnu.org
[2] *Trac* es una plataforma para la gestión de proyectos disponible en:http://trac.edgewall.org/

dades pueden ejecutarse más de una vez en relación a esquemas de “maduración” vinculados a los productos intermedios y finales.

En esta segunda versión la noción de comunidad se redimensiona a la exploración de nuevas formas de trabajo basadas en la expansión del conocimiento y la conciencia de los procesos de enseñanza-aprendizaje en la práctica de desarrollo. En este sentido, se toman experiencias relevantes de las comunidades de software libre para integrarlas a la estructura planteada.

Desde el año 2007 la primera versión de la metodología de desarrollo de software libre ha sido utilizada por varios equipos de trabajo de la Fundación Centro Nacional de Desarrollo e Investigación en Tecnologías Libres, logrando con su uso la rápida adopción de los conceptos inherentes al Software Libre y la pronta respuesta a necesidades de software crítico vinculados a proyectos tales como la Autoridad Certificación Raíz Venezolana, el Mapa Industrial de Venezuela, el Sistema de Planificación Estratégica Situacional para la Administración Pública Venezolana y la masificación de la Firma Electrónica.



METODOLOGÍA PARA EL DESARROLLO COLABORATIVO DE SOFTWARE LIBRE

J. Alvarez y V. Bravo

Fundación Centro Nacional de Desarrollo e Investigación en Tecnologías Libres

La propuesta metodológica que se presenta en este documento constituye una segunda versión de la Metodología para el Desarrollo Colaborativo de Software Libre [1] publicada por la Fundación Centro Nacional de Desarrollo e Investigación en Tecnologías Libres (CENDITEL) en el año 2007. Esta segunda versión se fundamenta en el concepto de práctica virtuosa definido por MacIntyre en el libro "Tras la Virtud" [2], en el cual una práctica se define como una actividad humana cooperativa, socialmente establecida, mediante la cual se busca la mejora continua de lo producido en ésta, lo que implica un proceso de cultivo de conocimiento asociado a la práctica en pro de la consecución de la excelencia en la ejecución de la misma. En específico, para esta segunda versión de la metodología se toma del concepto de práctica algunos aspectos relacionados al cultivo del conocimiento y al sentido social de la práctica, en base a los cuales se plantea esta propuesta metodológica que pretende servir de eje orientador hacia la búsqueda de una práctica virtuosa en el desarrollo de software libre.

En lo que se refiere al cultivo de conocimiento, se busca promover un proceso de enseñanza-aprendizaje en torno a la práctica de desarrollo de software, lo cual implica que los involucrados en ésta lleven a cabo actividades que faciliten tanto la enseñanza como el aprendizaje de conocimientos y experiencias que posean

sus integrantes en relación a dicha práctica. Este proceso contribuye así a la mejora continua de la práctica de desarrollo en base a las experiencias adquiridas en este ámbito, así como en base a modelos y patrones de excelencia que las comunidades de software libre han ido cultivando en el transcurrir del tiempo. Para ello, esta segunda versión se fundamenta en:

- Propuestas de mejoras a la primera versión de la metodología de desarrollo, planteadas como resultado de la aplicación de esta metodología en proyectos de software libre de CENDITEL.
- Planteamiento de una segunda versión del Proceso de Aseguramiento de Calidad en el Desarrollo de Software Libre [3], publicado por CENDITEL en el año 2013. Este proceso tiene como objetivo mejorar la calidad del software, tanto a nivel de la práctica de desarrollo de éste como a nivel de las aplicaciones desarrolladas.
- Modelos o patrones de excelencia que se han ido cultivando en el tiempo dentro de las comunidades y organizaciones que desarrollan software (tanto a nivel propietario como libre), los cuales permiten ejecutar actividades de la práctica de desarrollo de una mejor manera, contribuyendo a su mejora continua y al incremento de calidad de las aplicaciones desarrolladas en ella.

En lo que respecta al sentido social de la práctica, se tiene como propósito consolidar una práctica de desarrollo en la cual se otorgue especial énfasis a la documentación del software, a fin de facilitar procesos de apropiación que apunten no sólo al uso sino también al mantenimiento y mejora del software por parte de usuarios e interesados. En este caso, la documentación del software constituye un elemento fundamental para facilitar, a partir del software publicado, su mejora o la construcción de otras aplicaciones. Así, el desarrollo de software libre y la generación de su documentación juegan un papel importante para el alcance de la Soberanía e Independencia Tecnológica de la Nación, tanto en lo que se refiere a la generación de conocimiento con fines sociales como en lo que respecta al desarrollo pertinente y uso de aplicaciones para áreas de interés nacional, como por ejemplo, el sector socio-productivo, educativo y de salud, que puedan ser mejoradas y mantenidas en nuestro país sin incurrir en el pago de licencias y en la contribución al monopolio del conocimiento, implícito en el software que usualmente se utiliza en estas áreas.

Por consiguiente, para orientar esta segunda versión de la metodología hacia una práctica de cultivo de conocimiento que involucre no sólo a los practicantes del desarrollo de software sino también a los usuarios e interesados en las aplicaciones desarrolladas, se plantea un conjunto de recomendaciones en torno a las actividades descritas en los procesos de la metodología. Estas recomendaciones proponen una mejor forma de realizar la práctica de desarrollo en base a modelos y patrones de excelencia, así como en base a experiencias adquiridas en la práctica de desarrollo que se da en CENDITEL.

Los procesos que componen la segunda versión de la metodología son los siguientes: Conceptualización de Proyectos de Software Libre, Administración de Proyectos de Software Libre y Construcción de Aplicaciones de Software Libre. Los procesos de la metodología y sus relaciones básicas se presentan en la Figura 1.

Tal como se muestra en la Figura 1, los tres procesos se integran entre sí a través de los principales productos que se generan en cada uno de éstos. Por ejemplo, el alcance del proyecto generado en el proceso de Conceptualización es suministrado al proceso de Administración para generar allí el plan del proyecto, en el que se indican las funcionalidades a desarrollar en el proceso de construcción.

Para agilizar la práctica de desarrollo en base a la segunda versión de la metodología, así como para apoyar las actividades de documentación del software planteadas en ésta, en CENDITEL se trabaja en una segunda versión del **complemento**[1] **desarrollado para la plataforma *Trac*** en dicha Fundación, en el año 2011, con la finalidad de brindar apoyo a la ejecución de los procesos de la metodología. Cabe destacar que este complemento junto a las demás herramientas que brinda la plataforma *Trac*, permiten llevar a cabo la mayoría de las actividades que componen la práctica de desarrollo planteada en esta metodología.

[1] Este complemento o plugin puede ser descargado en: https://calidad-sl.cenditel.gob.ve/trac

También, es importante señalar que la propuesta metodológica que se presenta no se encuentra atada a herramientas específicas de apoyo a la práctica de desarrollo, por lo que los procesos planteados pueden apoyarse en cualquier herramienta libre que facilite su ejecución y documentación, que a su vez favorezca el trabajo colaborativo.

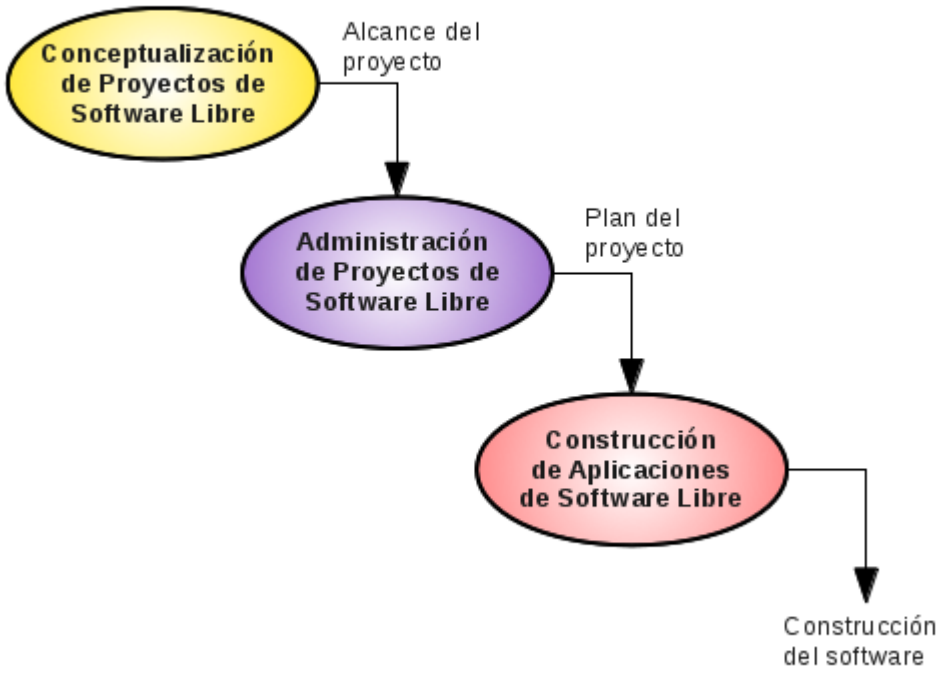


Figura 1 Procesos de la metodología y sus relaciones básicas

1.1. Proceso de Conceptualización de Proyectos de Software Libre

En este proceso se recopila y analiza información concerniente a los procesos que se requieren automatizar en una aplicación de software, con el objetivo de comprender el dominio de la aplicación a desarrollar así como los problemas o necesidades de los usuarios en relación a dichos procesos, todo ello con la finalidad de plantear una propuesta de desarrollo de software acorde a los requerimientos de los usuarios.

A continuación, en la Figura 2 se presenta el diagrama de flujo de trabajo correspondiente a la secuencia de ejecución de las actividades contempladas para el proceso de Conceptualización de Proyectos de Software Libre. Luego, en la Tabla 1 se describen las tareas correspondientes a cada una de las actividades indicadas en el flujo respectivo.

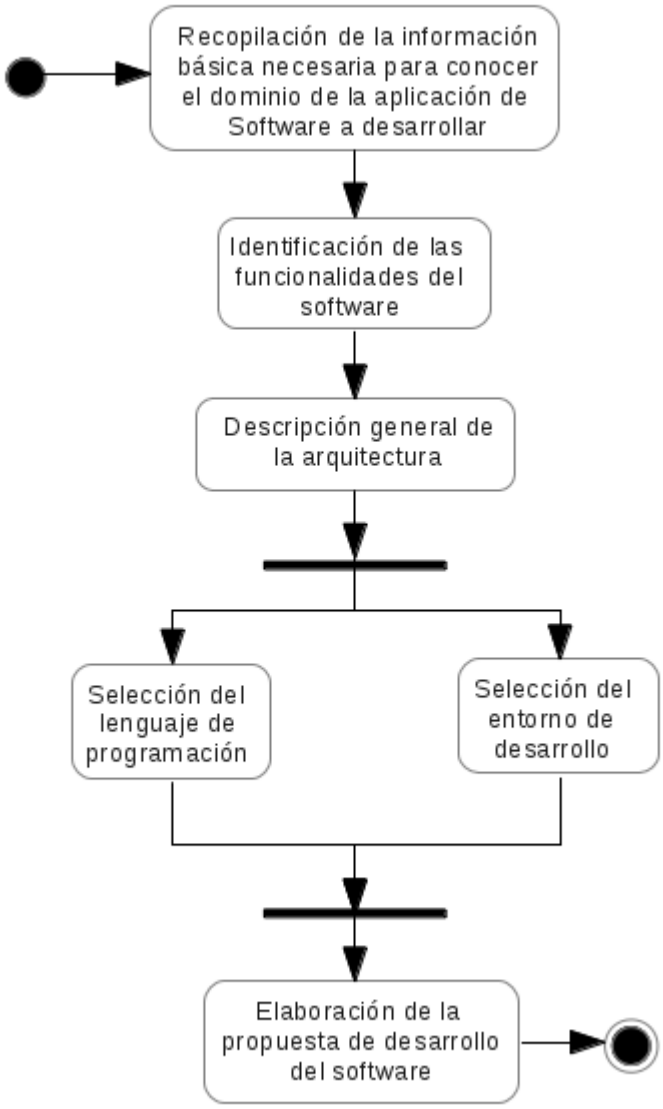


Figura 2 Flujo de trabajo del proceso de Conceptualización de Proyectos de Software Libre

Tabla 1: Tareas que integran las actividades asociadas al proceso de Conceptualización de Proyectos de Software Libre

<table>
<tr><th colspan="2">Actividad: Recopilación de la información básica necesaria para conocer el dominio de la aplicación de software a desarrollar</th></tr>
<tr><td rowspan="4">Tarea: Identificar los procesos que se quieren automatizar, así como las problemáticas y/o necesidades en torno a éstos.</td><td>Recomendaciones:
Para identificar los problemas y/o necesidades en torno a los procesos a automatizar, se requiere realizar reuniones entre los miembros del Equipo de Desarrollo y los usuarios e interesados en el software a desarrollar.
En las conversaciones con los usuarios e interesados se debe recopilar información sobre los procesos a automatizar, en base a la cual se pueda modelar tales procesos con el objetivo de entenderlos en función de las actividades que los integran.
Es importante que la información que se recopilará sobre los procesos a automatizar, contribuya a la identificación de requerimientos no funcionales que deba cumplir el software a desarrollar, como por ejemplo, información respectiva al número aproximado de personas que usaran el software, así como su frecuencia de uso. Este tipo de información representa un insumo básico para definir el tipo de arquitectura del software.</td></tr>
<tr><td>Herramientas:
Para los casos en los que se requiera realizar reuniones entre personas ubicadas en distintos espacios geográficos, se pueden utilizar herramientas de comunicación como: Dimdim-OpenSource, Openmeetings, entre otras.
Las minutas que se generen en las conversaciones con los usuarios y/o interesados pueden registrarse en la plataforma de desarrollo del proyecto.</td></tr>
<tr><td>Productos:
Minutas.</td></tr>
<tr><td>Responsables:
Líder del proyecto, analistas.</td></tr>
<tr><td colspan="2">La tabla continúa en la siguiente página</td></tr>
</table>

<table>
<tr><td></td><td>Colaboradores:
Usuarios, interesados y cualquier miembro del equipo de desarrollo.</td></tr>
<tr><td colspan="2">Actividad: Identificación de las funcionalidades del software</td></tr>
<tr><td rowspan="5">Tarea: Elaborar los diagramas de procesos a automatizar.</td><td>Recomendaciones:
Un diagrama de procesos constituye una representación gráfica que capta la estructura y la dinámica de un proceso, lo cual permite describir éste especificando los datos (entrada y salida), las actividades, los roles y las reglas que regulan dicho proceso.
Cada diagrama de proceso debe contener los elementos que describen el proceso, a saber: entradas, productos (salidas), recursos, reglas, objetivos y actores.
Se pueden utilizar los diagramas caja negra para representar los procesos [4].</td></tr>
<tr><td>Herramientas:
Para elaborar los diagramas de procesos se pueden utilizar herramientas gráficas como Dia, entre otras.
Los diagramas de procesos pueden registrarse en el wiki de “Análisis del dominio de la aplicación”, incluido en el complemento desarrollado para la plataforma Trac.</td></tr>
<tr><td>Productos:
Diagramas de procesos.</td></tr>
<tr><td>Responsables:
Analistas.</td></tr>
<tr><td>Colaboradores:
Usuarios.</td></tr>
<tr><td colspan="2">La tabla continúa en la siguiente página</td></tr>
</table>

<table>
<tr><td rowspan="5">Tarea: Elaborar el diagrama de relación entre los procesos a automatizar.</td><td>Recomendaciones:
En este diagrama se debe representar la relación entre los procesos en términos de los productos que se generan en cada proceso y que se requieren como insumos (entradas y/o recursos) en otros procesos [4].</td></tr>
<tr><td>Herramientas:
Para elaborar el diagrama de relación entre procesos se pueden utilizar herramientas gráficas como: Dia, IDEFO, Bonita, entre otras.
El diagrama de relación entre procesos puede registrarse en el wiki de “Análisis del dominio de la aplicación”, incluido en el complemento desarrollado para la plataforma Trac.</td></tr>
<tr><td>Producto:
Diagrama de relación entre procesos.</td></tr>
<tr><td>Responsables:
Analistas.</td></tr>
<tr><td>Colaboradores:
Usuarios.</td></tr>
<tr><td>Tarea: Elaborar los diagramas de actividades correspondientes a cada proceso a automatizar.</td><td>Recomendaciones:
A fin de desarrollar un software que aporte mejoras en la ejecución de los procesos a automatizar, se recomienda analizar los diagramas de actividades [4] de cada uno de ellos, con el objetivo de identificar inconsistencias y conflictos en el flujo de ejecución de éstos, así como identificar si es necesario agregar o eliminar actividades a los mismos. De este análisis se pueden generar nuevos diagramas de actividades en los que se propongan mejoras en la ejecución de los respectivos procesos.</td></tr>
<tr><td colspan="2">La tabla continúa en la siguiente página</td></tr>
</table>

	Herramientas: ▪ Para elaborar los diagramas de actividades se pueden utilizar herramientas gráficas como: Dia, *Umbrello*, Bonita, *CASEUML*, *ArgoUML*, *BOUML*, entre otras. ▪ Los diagramas de actividades por proceso pueden registrarse en el *wiki* de "Análisis del dominio de la aplicación", incluido en el complemento desarrollado para la plataforma *Trac*.
	Productos: ▪ Diagramas de actividades por proceso.
	Responsables: ▪ Analistas.
	Colaboradores: ▪ Usuarios.
Tarea: Validar los diagramas de procesos y de actividades con los usuarios.	Recomendaciones: ▪ La validación de los diagramas de procesos y actividades por parte de los usuarios es determinante para el desarrollo de software, pues los usuarios son quienes conocen con detalle cada uno los procesos que se requiere automatizar, de allí la relevancia de que sean éstos quienes revisen los diagramas en los cuales se modelan estos procesos, y en base a los cuales se diseñará el software respectivo.
	Productos: ▪ Diagramas de procesos validados. ▪ Diagrama de relación entre procesos validado. ▪ Diagramas de actividades validados.
	Responsables: ▪ Analistas y usuarios.
La tabla continúa en la siguiente página	

<table>
<tr><td rowspan="5">Tarea: Elaborar los diagramas de casos de uso en los que se represente el alcance del software, en base a las funcionalidades generales del mismo.</td><td>Recomendaciones:
Las funcionalidades que debe brindar el software se identifican en base a los diagramas de procesos y actividades.
Para facilitar la lectura de los diagramas de casos de uso [4] se recomienda no superar más de tres niveles de relación entre casos de uso (ya sean relaciones de inclusión, extensión o generalización).</td></tr>
<tr><td>Herramientas:
Para elaborar los diagramas de casos de uso se pueden utilizar herramientas gráficas como: Dia, Umbrello, Bonita, CASEUML, ArgoUML, BOUML, entre otras.
En el caso de CENDITEL se plantea utilizar para elaborar los diagramas de casos de uso la herramienta Platuml, incluida en el complemento desarrollado para la plataforma Trac.</td></tr>
<tr><td>Productos:
Diagramas de casos de uso (alcance del software).</td></tr>
<tr><td>Responsables:
Analistas.</td></tr>
<tr><td>Colaboradores:
Cualquier miembro del equipo de desarrollo, usuarios.</td></tr>
<tr><td>Tarea: Identificar potenciales actores colaboradores en el desarrollo del software.</td><td>Recomendaciones:
A fin de fomentar el trabajo colaborativo en torno al desarrollo de los proyectos de software, se considera pertinente identificar con qué actores se podría establecer dicho trabajo. Los actores colaboradores engloban tanto a los usuarios del software como actores que tengan conocimiento y experiencia en el desarrollo del tipo de software que se propone. En el caso de los usuarios es importante fomentar la participación de éstos en la práctica de desarrollo del software, pues de esta manera el proceso de apropiación del software podrá darse con mayor facilidad.
A partir de la identificación de los potenciales actores colaboradores, se recomienda al equipo de desarrollo buscar un tipo de articulación que permita fomentar este trabajo colaborativo.</td></tr>
<tr><td colspan="2" align="center">La tabla continúa en la siguiente página</td></tr>
</table>

<table>
<tr><td rowspan="3"></td><td>Producto:
Potenciales actores colaboradores.</td></tr>
<tr><td>Responsables:
Líder del proyecto.</td></tr>
<tr><td>Colaboradores:
Equipo de desarrollo, usuarios.</td></tr>
<tr><td colspan="2">Actividad: Descripción general de la arquitectura</td></tr>
<tr><td>Tarea: Describir a grandes rasgos el tipo de arquitectura del software a desarrollar.</td><td>Recomendaciones:
Al momento de definir la arquitectura del software, es decir, sus componentes y la interacción (comunicación) entre éstos, se requiere tener en cuenta no sólo las funcionalidades del software y las limitaciones tecnológicas existentes, sino también los atributos de calidad asociados al software, ya que la arquitectura que se defina puede inhibir o facilitar el cumplimiento de dichos atributos [5]. Por ejemplo, el atributo de calidad respectivo al desempeño del software depende de los componentes que se definan para éste y de su ubicación en los procesadores, así como de los caminos de comunicación entre dichos componentes, etc.

Otro elemento a considerar para la definición de la arquitectura es la disponibilidad de componentes que faciliten generar bitácoras de funcionamiento con niveles de detalle configurables, las cuales son de gran utilidad para la detección y corrección de errores. Similarmente, otro elemento de interés para dicha definición es la disponibilidad de funcionalidades que posibiliten la realización de pruebas de perfilamiento, las cuales ayudan a detectar componentes para optimización de desempeño.
Para el desarrollo de aplicaciones de software complejas y de gran escala se recomienda utilizar patrones y/o estilos arquitectónicos, así como patrones de diseño, debido a que éstos permiten mejorar la calidad de las aplicaciones. Para la selección de patrones y estilos es necesario tener presente que éstos pueden facilitar el cumplimiento de ciertos atributos de calidad en un software, pero a su vez pueden inhibir el cumplimiento de otros atributos. Por ejemplo, el patrón arquitectónico Modelo-Vista-Controlador facilita el cumplimiento de atributos de calidad asociados a la funcionalidad y la mantenibilidad, pero dificulta el cumplimiento de atributos de calidad asociados al desempeño y a la portabilidad del software [6].</td></tr>
<tr><td colspan="2">La tabla continúa en la siguiente página</td></tr>
</table>

<table>
<tr><td></td><td>

Es importante que se estudien varias alternativas de la arquitectura, de modo que se pueda seleccionar de éstas aquella que permita cumplir en mayor medida con los requerimientos funcionales y los atributos de calidad establecidos para el software, que facilite futuras modificaciones al mismo y que sea factible conforme las limitaciones tecnológicas que puedan existir.
Para describir de manera general la arquitectura del software se pueden utilizar diagramas sencillos, donde se representen a grandes rasgos los componentes del software. Los diagramas más detallados de la arquitectura, como por ejemplo, diagramas de clase, diagramas de secuencia, pueden ser elaborados en la fase de Análisis y Diseño, contenida en el proceso de Construcción de Aplicaciones de Software Libre, pues en esta etapa se tiene mayor conocimiento de los componentes del software, así como de sus interacciones.

</td></tr>
<tr><td></td><td>Herramientas:

El diagrama de la arquitectura de software puede registrarse en el wiki de “Arquitectura del software”, incluido en el componente desarrollado para la plataforma Trac.

</td></tr>
<tr><td></td><td>Productos:

Diagrama de la arquitectura del software.

</td></tr>
<tr><td></td><td>Responsable:

Arquitecto de software.

</td></tr>
<tr><td></td><td>Colaboradores:

Analistas, otros integrantes del equipo de desarrollo.

</td></tr>
<tr><td>Tarea: Fundamentar el uso de la arquitectura seleccionada.</td><td>Recomendaciones:

A fin de hacer explicito las decisiones de diseño se sugiere elaborar un documento en el que se fundamenten las razones a las cuales obedece la selección de la arquitectura del software, en función de los criterios utilizados. Ello facilita no sólo el entender la arquitectura sino también futuros procesos de mantenimiento del software.

</td></tr>
<tr><td colspan="2">La tabla continúa en la siguiente página</td></tr>
</table>

<table>
<tr><td></td><td>Herramienta:
El documento de fundamentación de diseño arquitectónico puede registrarse en el wiki de “Arquitectura del software”, incluido en el componente desarrollado para la plataforma Trac.</td></tr>
<tr><td></td><td>Producto:
Documento de fundamentación de diseño arquitectónico.</td></tr>
<tr><td></td><td>Responsable:
Arquitecto de software.</td></tr>
<tr><td></td><td>Colaboradores:
Analistas, otros integrantes del equipo de desarrollo.</td></tr>
<tr><td colspan="2">Actividad: Selección de lenguajes de programación</td></tr>
<tr><td>Tarea: Seleccionar el o los lenguajes de programación a utilizar en el desarrollo de software.</td><td>Recomendaciones:
Para seleccionar el o los lenguajes de programación es conveniente tener en cuenta [7]:
Portabilidad del software. Si el software a desarrollar debe poder operar en varios sistemas operativos, se requiere entonces de lenguajes de programación que permitan el ajuste del código fuente para que compile en estos sistemas.
Reutilización de código. Si se quiere hacer uso de código existente, por ejemplo, de bibliotecas para generar interfaces gráficas de usuario o establecer conexiones de red, se requiere de lenguajes de programación para los cuales estas bibliotecas brinden interfaz.
Acceso al hardware. Las aplicaciones de software que no requieren manipular componentes del hardware, como por ejemplo, un juego de vídeo, ameritan de lenguajes de programación de alto nivel, que facilitan el manejo del uso de la memoria automáticamente. Todo lo contrario a las aplicaciones que manipulan componentes del hardware, para las cuales se requiere lenguajes de bajo nivel como C.</td></tr>
<tr><td></td><td>Producto:
Documento de fundamentación de la elección de los lenguajes de programación a utilizar en el desarrollo.</td></tr>
<tr><td colspan="2">La tabla continúa en la siguiente página</td></tr>
</table>

<table>
<tr><td></td><td>Responsable:
Líder del proyecto.</td></tr>
<tr><td></td><td>Colaboradores:
Programadores y demás integrantes del equipo de desarrollo.</td></tr>
<tr><td>Tarea: Seleccionar un entorno de desarrollo (framework).</td><td>Recomendaciones:
Dado las bondades que ofrecen los entornos de desarrollo, se recomienda el uso de estas herramientas a fin de reutilizar librerías, funciones, y otras ventajas que éstas ofrecen en relación a la construcción de código. Existen numerosos entornos de desarrollo, entre ellos: Django, Ruby on Rails, Symfony, QT.
Para seleccionar el entorno de desarrollo se recomiendan algunos criterios [8] como:
Popularidad, es decir, el entorno de desarrollo debe ser reconocido entre la comunidad de usuarios, lo cual indica calidad y completitud de sus componentes.
Sostenibilidad, ello implica mantenimiento actualizado del entorno.
Soporte, implica la facilidad de encontrar respuestas a las preguntas que puedan surgir sobre el entorno de desarrollo. En este caso es necesario identificar los responsables de brindar dicho soporte, así como los medios que se utilicen para ello (listas de correro, IRC, entre otros).
Uso de patrones de diseño, lo cual permite la reutilización de código estándar para efectuar tareas particulares dentro del software a desarrollar.
Seguridad, el entorno de desarrollo debe ofrecer funciones que minimicen casos de vulnerabilidad en el software.</td></tr>
<tr><td colspan="2" align="center">La tabla continúa en la siguiente página</td></tr>
</table>

<table>
<tr><td></td><td>6. Documentación que facilite el uso del entorno de desarrollo.
7. Licencia de uso, este criterio es muy importante dado que la utilización de un entorno de desarrollo implica automáticamente la aplicación de la licencia de éste sobre el software desarrollado.
8. Prueba del entorno, ello permite constatar la aplicabilidad de éste al tipo de desarrollo a realizar.
9. Facilidad de aprendizaje, es decir, se debe requerir poco tiempo para aprender a utilizar el entorno de desarrollo.
10. Interoperabilidad, ello implica que las funciones y librerías que ofrece el entorno puedan interactuar con otros software.
▪ Además de los criterios mencionados para la selección del entorno debe considerarse también el tipo de arquitectura del software, ya que los entornos brindan elementos como los patrones de diseño, entre otros, que pueden ser determinantes para la arquitectura.</td></tr>
<tr><td></td><td>Producto:
▪ Entorno de desarrollo seleccionado y la fundamentación de dicha selección.</td></tr>
<tr><td></td><td>Responsable:
▪ Líder del proyecto.</td></tr>
<tr><td></td><td>Colaboradores:
▪ Equipo de desarrollo</td></tr>
<tr><td colspan="2">Actividad: Elaboración de la propuesta de desarrollo del software</td></tr>
<tr><td>Tarea: Elaborar la propuesta de desarrollo.</td><td>Recomendaciones:
▪ La propuesta de desarrollo debe contener información respectiva a la conceptualización del proyecto, por ejemplo, secciones de información referidas a:</td></tr>
<tr><td colspan="2">La tabla continúa en la siguiente página</td></tr>
</table>

	1. Necesidades y/o problemáticas que se abordarán con el software a desarrollar. 2. Solución que se propone (tipo de software). 3. Alcance del software. 4. Descripción general de la arquitectura. 5. Potenciales actores colaboradores en el desarrollo del software. 6. Metodología de desarrollo. 7. Plataforma de operación. 8. Plataforma de desarrollo, indicando el manejador de base de datos, los lenguajes de programación, el entorno de desarrollo, entre otros. 9. Licencias de código y documentación. ▪ En la sección "Alcance del software" se recomienda mostrar los diagramas de casos de uso diseñados para representar las funciones generales del software, ello facilita la comprensión del alcance del mismo.
	Herramienta: ▪ La información que debe contener esta propuesta puede registrarse en el *wiki* correspondiente a la "Propuesta de desarrollo", incluido en el complemento desarrollado para la plataforma *Trac*.
	Producto: ▪ Propuesta de desarrollo.
	Responsable: ▪ Líder del proyecto.
	Colaboradores: ▪ Analista, cualquier miembro del equipo de desarrollo.

Para contar con una documentación del software que pueda servir de ayuda a los procesos de apropiación del mismo y a la práctica de desarrollo, se requiere mantener actualizada la documentación generada en este proceso, así como la publicación de dicha documentación en la plataforma de desarrollo del proyecto y/o en su sitio web.

1.2. Proceso de Administración de Proyectos de Software Libre

En este proceso se realizan actividades de planificación, coordinación y seguimiento de las tareas del Equipo de Desarrollo, con el objetivo de lograr una buena ejecución de la práctica de desarrollo que tribute a la colaboración en la ejecución de la misma y a la apropiación del software. Para describir las actividades y tareas que se contemplan en el proceso de Administración de Proyectos de Software Libre se utiliza la misma estructura presentada en el proceso anterior. Ver Figura 3 y Tabla 2.

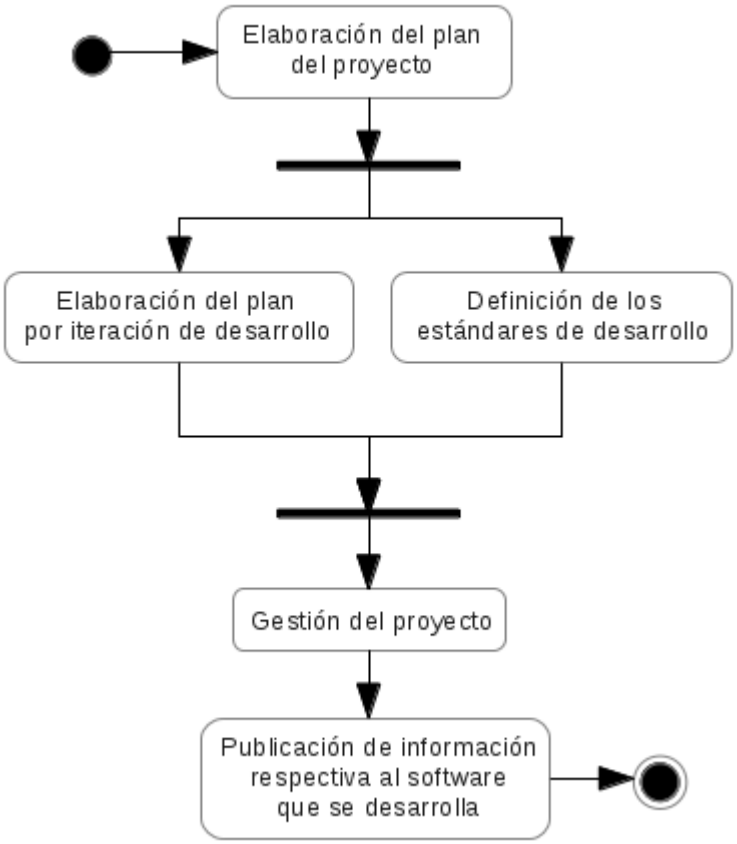


Figura 3 Flujo de trabajo del proceso de Administración de Proyectos de Software Libre

Tabla 2: Tareas que integran las actividades asociadas al proceso de Administración de Proyectos de Software Libre

Actividad: Elaboración del plan del proyecto	
Tarea: Definir y priorizar las funcionalidades del software.	Recomendaciones: ▪ Las funcionalidades generales del software se encuentran definidas en el alcance del mismo, sin embargo, al momento de priorizar las funcionalidades se recomienda estudiar con detalle las mismas. Dicha recomendación se plantea dado que en esta fase temprana del proyecto puede ser necesario definir nuevas funcionalidades o modificar las existentes. De ser así, es necesario actualizar el alcance del software en el documento de la propuesta de desarrollo. ▪ A fin de poder establecer con mayor claridad el tiempo de desarrollo del software, se recomienda desagregar aquellas funcionalidades que hayan sido definidas de forma general, es decir, aquellas funcionalidades que a su vez se compongan de otras. ▪ Para liberar prototipos del software que sean útiles a los usuarios, conforme a la urgencia de sus necesidades, se recomienda que los usuarios indiquen la prioridad con la que requieren las funcionalidades del software. ▪ La priorización de funcionalidades debe contener el listado de funcionalidades identificadas y la prioridad que los usuarios asignen a cada una de éstas.
	Herramientas: ▪ Las funcionalidades del software junto a su priorización por parte de los usuarios pueden registrarse en el *wiki* correspondiente al "Plan del proyecto", incluido en el componente desarrollado para la plataforma *Trac*.
	Producto: ▪ Priorización de funcionalidades por parte de los usuarios.
	Responsables: ▪ Usuarios y/o interesados, líder del proyecto.
La tabla continúa en la siguiente página	

	Colaboradores: ▪ Equipo de desarrollo.
Tarea: Definir el orden de dependencia entre las funcionalidades del software.	Recomendaciones: ▪ El establecer la dependencia entre funcionalidades, al igual que priorizarlas, permite una mejor toma de decisiones respecto al orden en que deberían desarrollarse, por lo cual estas tareas son fundamentales al momento de generar el plan del proyecto.
	Herramientas: ▪ La dependencia entre funcionalidades del software puede registrarse en el *wiki* correspondiente al "Plan del proyecto", incluido en el componente desarrollado para la plataforma *Trac*.
	Producto: ▪ Dependencia entre funcionalidades.
	Responsable: ▪ Líder del proyecto.
	Colaboradores: ▪ Equipo de desarrollo.
Tarea: Realizar un estudio sobre los riesgos de desarrollo del software.	Recomendaciones: ▪ Por riesgo se entiende todo aquello que pueda interferir o dificultar el desarrollo de funcionalidades del software, como por ejemplo, la falta de conocimiento y experiencia en el tipo de lenguaje a utilizar para el desarrollo. Todo riesgo debe ir asociado a una o más funcionalidades. ▪ El estudio de riesgos debe contener la definición y el impacto de los mismos, así como la prioridad para abordarlos y las acciones para prevenirlos.
La tabla continúa en la siguiente página	

	▪ La identificación de riesgos permite al equipo de desarrollo realizar una planificación concreta, ajustada a la dificultades que se puedan presentar durante el desarrollo del software, permitiendo así disminuir las diferencias entre lo planificado y lo ejecutado. ▪ Los riesgos deben ser priorizados en términos de su impacto en el desarrollo de las funcionalidades del software. ▪ Es importante que se definan acciones preventivas que puedan llevarse a cabo en función de evitar la ocurrencia de los riesgos identificados. En el caso de riesgos asociados a la complejidad del desarrollo de funcionalidades o métodos del software, se recomienda como acción preventiva la puesta en práctica de "Pruebas de Concepto". Estas pruebas permiten implementar, de manera resumida e incompleta, un método, función o idea que pueda ser o parecer compleja, con el propósito de verificar si es posible su desarrollo [9], así como determinar los aspectos a considerar durante el desarrollo del software relacionados a la implementación completa de dicho método, función o idea.
	Herramienta: ▪ El estudio de riesgos puede registrarse en el *wiki* correspondiente al "Plan del proyecto", incluido en el componente desarrollado para la plataforma *Trac*.
	Producto: ▪ Estudio de riesgos.
	Responsable: ▪ Líder del proyecto.
	Colaboradores: ▪ Equipo de desarrollo.
La tabla continúa en la siguiente página	

Tarea: Elaborar el plan del proyecto.	Recomendaciones: ▪ La priorización de funcionalidades y riesgos, junto al orden de dependencia entre funcionalidades, constituyen los fundamentos sobre los cuales se deberían tomar decisiones con respecto a la prioridad de desarrollo de las funcionalidades del software, por tanto, son la base para elaborar el plan de desarrollo de un proyecto de software. Esta manera de planificar permite ir desarrollando prototipos que sean útiles a los usuarios conforme sus necesidades, teniendo en cuenta para ello el orden de dependencia entre las funcionalidades del software y la necesidad de abordar los riesgos más importantes del desarrollo en etapas tempranas de éste. ▪ En base a la consideración de los tres fundamentos indicados en el párrafo anterior, se ha planteado un conjunto de formulas que permitan calcular la priorización de desarrollo de cada funcionalidad del software, a saber: $$IF_j = (\sum_{i=1,\ldots,n} VR_{i\ asociados\ a\ Fj}) * PR + VF_j * PF$$ donde IF_j representa la importancia de la funcionalidad j según la prioridad que le da el usuario y según la prioridad de los riesgos asociados a dicha funcionalidad; VR_i representa el valor de prioridad para abordar el riego i asociado a la funcionalidad j; PR representa el peso asociado al factor riesgo; VF_j representa el valor de prioridad asignado por los usuarios a la funcionalidad j; PF representa el peso asociado al factor funcionalidad. El PR y el PF son factores utilizados para realizar el cálculo ponderado de priorización de desarrollo. A cada uno de estos factores se les debe asignar valores comprendidos entre 0 y 1. El valor asignado a cada factor dependerá de la importancia que adquieran éstos para el desarrollo de la aplicación. $$PDF_j = IF_j + \sum IF$$ donde PDF_j representa la prioridad de desarrollo de la funcionalidad j ; IF representa la importancia de una funcionalidad que se construye a partir de la funcionalidad j. El cálculo de la Prioridad de Desarrollo de las Funcionalidades (PDF) permite crear un orden de prioridad para la construcción de las funcionalidades del software. Las funcionalidades para las cuales se obtengan los valores más altos de PDF deben ser construidas en las primeras iteraciones del proyecto, dado que éstas corresponden a las funcionalidades de mayor prioridad para los usuarios, que tienen asociados riesgos de mayor prioridad para ser abordados, y a partir de las cuales se pueden construir otras funcionalidades del software. En este sentido, tales funcionalidades pueden ser consideradas como el núcleo de la aplicación a desarrollar.

La tabla continúa en la siguiente página

<table>
<tr><td></td><td>

- El documento del plan del proyecto debe contener información respectiva a:
  1. Priorización y orden de dependencia de las funcionalidades del software.
  2. Estudio de riesgos (priorización y acciones preventivas).
  3. Priorización de desarrollo de cada funcionalidad.
  4. Cronograma de desarrollo del proyecto.
- En el cronograma de desarrollo del proyecto se debe especificar el número de iteraciones a realizar, indicando por cada iteración las funcionalidades a desarrollar y las fechas de inicio y fin de cada iteración. La toma de decisiones sobre cuáles funcionalidades desarrollar por iteración, se realiza en base al cálculo de la prioridad de desarrollo de cada funcionalidad.

</td></tr>
<tr><td></td><td>

Herramienta:

- La información que debe contener el plan del proyecto puede registrarse en el *wiki* correspondiente al “Plan del proyecto”, incluido en el componente desarrollado para la plataforma *Trac*.

</td></tr>
<tr><td></td><td>

Producto:

- Plan del proyecto.

</td></tr>
<tr><td></td><td>

Responsable:

- Líder del proyecto.

</td></tr>
<tr><td></td><td>

Colaboradores:

- Equipo de desarrollo.

</td></tr>
<tr><td colspan="2">Actividad: Definición de los estándares de desarrollo</td></tr>
<tr><td>Tarea: Definir los estándares de desarrollo.</td><td>

Recomendaciones:

- Los estándares de desarrollo no sólo abarcan estándares de codificación, incluyen también estándares para interfaz gráfica, para base de datos, entre otros.

</td></tr>
<tr><td colspan="2">La tabla continúa en la siguiente página</td></tr>
</table>

<table>
<tr><td></td><td>

- Los estándares de codificación permiten la lectura rápida y simple del código, facilitando así el trabajo en conjunto, por lo cual la utilización de este tipo de estándares es fundamental tanto para el trabajo colaborativo como para los procesos de mejoras posteriores que se realicen al software. Existen numerosos estándares de codificación según el tipo de lenguaje, por ejemplo: Guía de estilo para código *Python* [10], Guía de estilo de programación para código PHP [11], entre otros.

</td></tr>
<tr><td></td><td>

Herramienta:

- Los estándares que se definan para el proyecto pueden registrarse en el *wiki* correspondiente a “Estándares de desarrollo”, incluido en el componente desarrollado para la plataforma *Trac*.

</td></tr>
<tr><td></td><td>

Producto:

- Estándares de desarrollo.

</td></tr>
<tr><td></td><td>

Responsable:

- Líder del proyecto.

</td></tr>
<tr><td></td><td>

Colaboradores:

- Equipo de desarrollo.

</td></tr>
<tr><td colspan="2">Actividad: Elaboración del plan por iteración de desarrollo</td></tr>
<tr><td>Tarea: Elaborar el plan para la iteración de desarrollo actual.</td><td>

Recomendaciones:

- En el plan de la iteración se indican las tareas a realizar para la construcción de las funcionalidades correspondientes a una iteración, según se plantea en el plan del proyecto, indicando por cada tarea los responsables y el tiempo de entrega de los productos a obtener en las mismas.
- Para cada una de las iteraciones indicadas en el plan del proyecto se debe generar una planificación de tareas.

</td></tr>
<tr><td colspan="2">La tabla continúa en la siguiente página</td></tr>
</table>

	Herramientas: ▪ Existen varias herramientas para la planificación de tareas, entre ellas: *Planner*, *XPTraker*, los sistemas de asignación de tareas incluidos en plataformas de desarrollo de software como *Gforge*, *SourceForge*, *Trac*, entre otras. ▪ En el caso de CENDITEL se propone hacer uso del sistema de control de incidentes que ofrece la plataforma *Trac*, a fin de utilizarlo para la definición y asignación de tareas respectivas al desarrollo de funcionalidades del software.
	Producto: ▪ Plan de la iteración actual.
	Responsable: ▪ Líder del proyecto.
	Colaboradores: ▪ Equipo de desarrollo.
Actividad: Gestión del proyecto	
Tarea: Instalar una plataforma de desarrollo para gestionar el proyecto.	Recomendaciones: ▪ Existen varias plataformas informáticas de apoyo a la práctica de desarrollo de software, entre ellas: *XPTraker*, *Gforge*, *SourceForge*, *Trac*, *Redmine*. La plataforma *Trac* es la herramienta utilizada en CENDITEL para apoyar la práctica de desarrollo de software. ▪ En el caso de la herramienta *Trac* es importante destacar que CENDITEL ha desarrollado un complemento para dicha plataforma, el cual sirve de apoyo a los procesos planteados en esta metodología.
	Producto: ▪ Plataforma de desarrollo del proyecto que contiene información respectiva a la práctica de desarrollo del software.

La tabla continúa en la siguiente página

<table>
<tr><td></td><td>Responsable:
Líder del proyecto.</td></tr>
<tr><td></td><td>Colaboradores:
Equipo de desarrollo.</td></tr>
<tr><td rowspan="6">Tarea: Realizar reuniones periódicas entre el equipo de desarrollo.</td><td>Recomendaciones:
Se sugiere tener reuniones semanales para discutir asuntos del proyecto que tributen a la mejora continua de la práctica de desarrollo y del software respectivo. Es importante que estas reuniones sirvan como medio para generar un proceso de enseñanza-aprendizaje entre los integrantes del equipo de desarrollo, pues en ellas se pueden exponer las dificultades que se presenten al equipo durante el desarrollo del software, así como las lecciones aprendidas al momento de abordar dichas dificultades. De igual forma, en estas reuniones se discuten los avances del proyecto, y a su vez permiten al equipo de desarrollo tener una visión general del mismo.
Se sugiere reuniones cortas, y en caso de considerarse necesario, según los asuntos y acuerdos discutidos en las reuniones, se sugiere elaborar minutas de éstas.</td></tr>
<tr><td>Herramientas:
Las minutas que se generen pueden registrarse en la plataforma de desarrollo del proyecto.
Para los casos en los que se requiera realizar reuniones entre personas ubicadas en distintos espacios geográficos se pueden utilizar herramientas de comunicación como: Hangout, Skype, entre otras.</td></tr>
<tr><td>Productos:
Minutas.</td></tr>
<tr><td>Responsable:
Líder del proyecto.</td></tr>
<tr><td>Colaboradores:
Equipo de desarrollo.</td></tr>
<tr><td colspan="2" align="center">La tabla continúa en la siguiente página</td></tr>
</table>

<table>
<tr><td rowspan="3">Tarea: Realizar seguimiento de las tareas asignadas al equipo de desarrollo.</td><td>Recomendaciones:
Se sugiere que el seguimiento de las tareas incluya la verificación del cumplimiento de estándares, lo cual constituye un factor muy importante para facilitar tanto el desarrollo colaborativo como las modificaciones o agregados que se realicen al software a futuro.
En el caso de personas que trabajen a distancia se recomienda establecer una periodicidad mínima para presentación de informes, para cargar novedades en el sistema de control de versiones y el sistema de control de incidentes, así como reuniones periódicas a través de medios de comunicación, en las cuales se pueda discutir asuntos relacionados a los resultados de las tareas asignadas.</td></tr>
<tr><td>Herramientas:
Para realizar seguimiento de tareas se pueden utilizar herramientas para gestión de proyectos. Por ejemplo, el Trac cuenta con un sistema de incidentes (sistema de tickets) y manejo de hitos que pueden ser utilizados para seguimiento de tareas.</td></tr>
<tr><td>Responsable:
Líder del proyecto.</td></tr>
<tr><td>Tarea: Gestionar las incidencias reportados sobre el software.</td><td>Recomendaciones:
Para realizar el seguimiento y control de las incidencias que se puedan reportar, es decir, errores y solicitudes, se recomienda contar con un sistema para gestión de incidentes (Bug Tracking System) que permita llevar el registro de éstas, así como de su priorización para ser atendidas.
Por otro lado, para el reporte de incidencias por parte de los usuarios se recomienda contar con mecanismos sencillos para efectuar dichos reportes. Y en búsqueda de una mejor organización del trabajo colaborativo en torno al desarrollo del software, se sugiere colocar dichos mecanismos de reporte en el sitio web del proyecto.
Para facilitar la gestión en la corrección de los errores, se recomienda que el líder del proyecto sea quien gestione los errores para su respectiva corrección entre los miembros del equipo de desarrollo. Para facilitar esta tarea se podría programar el sistema de gestión de incidentes, a fin de que éste se encargue de enviar notificaciones sobre reporte de errores a la dirección que sea pertinente en el equipo de desarrollo.</td></tr>
<tr><td colspan="2" align="center">La tabla continúa en la siguiente página</td></tr>
</table>

<table>
<tr><td></td><td>Herramientas:
Existen muchas herramientas para la gestión de proyectos de software que incluyen sistemas para la gestión de errores, entre ellas se encuentran: Trac, Redmine, Open Atrium, Proyect-Open.
Entre las herramientas para gestión de errores se encuentran: Bugzilla, Mantis, Request Tracker, Eventum, entre otras.
En el caso de CENDITEL, se plantea utilizar el sistema de incidentes que incluye la plataforma Trac para realizar el seguimiento y control de errores en los proyectos que se desarrollan.</td></tr>
<tr><td></td><td>Producto:
Registro del seguimiento y control de los errores reportados.</td></tr>
<tr><td></td><td>Responsables:
Equipo de desarrollo.</td></tr>
<tr><td></td><td>Colaboradores:
Usuarios.</td></tr>
<tr><td colspan="2">Actividad: Construcción del sitio web del proyecto</td></tr>
<tr><td>Tarea: Construir un sitio web para la publicación de información respectiva al proyecto.</td><td>Recomendaciones:
Para todo proyecto de software libre es fundamental contar con un sitio web de acceso público (páginas o repositorio), que permita conformar una cartelera informativa del proyecto. Para ello se requiere de herramientas que faciliten mayores posibilidades en cuanto a estilo visual y elementos dinámicos en las páginas. Es importante que el sitio web permita el acceso a las versiones del software y a cualquier otra documentación del mismo que se considere pertinente, tanto para facilitar la apropiación del software como para promover la colaboración en torno a su desarrollo. En relación a las versiones del software deben publicarse también las versiones de prueba, pues ello facilita que personas ajenas al equipo de desarrollo puedan colaborar en la fase de pruebas, a través del reporte de los errores encontrados en el software.</td></tr>
<tr><td colspan="2">La tabla continúa en la siguiente página</td></tr>
</table>

	▪ Entre la información a publicar sobre el software, es fundamental colocar información de contacto y/o los mecanismos de comunicación con el equipo de desarrollo, a fin de que éstos puedan ser contactados para informar sobre el proyecto. De igual forma, es importante que a través del sitio de publicación del proyecto los usuarios puedan reportar errores sobre el software. ▪ Se recomienda que en el sitio web del proyecto se coloque un enlace a la plataforma de desarrollo del software, a fin de que las personas interesadas en el proyecto puedan tener acceso a toda la documentación que se genere.
	Herramienta: ▪ En el caso de CENDITEL, por lo general, se utilizan los *blog* como sitios web de los proyectos de desarrollo.
	Producto: ▪ Sitio web del proyecto con información sobre éste.
	Responsables: ▪ Líder del proyecto y miembros del equipo de desarrollo.

Para contar con una documentación del software que pueda servir de ayuda a los procesos de apropiación de éste y a la práctica de desarrollo, se requiere mantener actualizada la documentación generada en este proceso, así como la publicación de la documentación en la plataforma de desarrollo del proyecto y/o en su sitio web.

1.3. Proceso de Construcción de Aplicaciones de Software Libre

Una vez elaborado el plan del proyecto se prosigue a llevar a cabo las iteraciones planificadas. En cada iteración se realizan un conjunto de actividades que conforman el proceso de Construcción de la Aplicación de Software Libre. Este conjunto de actividades se agrupan en las siguientes fases: Especificación de Requerimientos, Análisis y Diseño, Codificación, Pruebas y Liberación.

En la Figura 4 se presentan las fases que componen el proceso de construcción y las relaciones entre éstas.

Figura 4 Gráfico de relación entre las fases que componen el proceso de Construcción de Aplicaciones de Software Libre

En la metodología la construcción de aplicaciones de software se da de manera incremental, en tal forma que en cada iteración[2] planificada se construye un número específico de funcionalidades, a partir de las cuales se obtiene una versión o prototipo de prueba (versión beta) que se entrega a los usuarios para su validación. Los errores reportados por los usuarios son corregidos por el equipo de desarrollo, obteniendo así una versión estable del software. La construcción de una versión del software requiere llevar a cabo las fases indicadas en la Figura 4, o por lo menos la mayoría de éstas. Por lo general, una iteración de desarrollo comienza por la fase de Especificación de Requerimientos y culmina en la fase de Liberación.

En los proyectos de software los cambios en los requerimientos suelen ocurrir con frecuencia, así como las actualizaciones en su documentación, razón por la cual se plantea en la Figura 4 un proceso de construcción lo bastante flexible, en el cual se puede pasar de una fase a otra sin importar la secuencia entre éstas.

Cabe destacar que para contar con una documentación del software que pueda servir de ayuda a los procesos de apropiación de éste, así como a la práctica de desarrollo, se requiere mantener actualizada y publicada en la plataforma de desarrollo del proyecto y/o en su sitio web, la documentación y el código generado en las fases del proceso de Construcción de Aplicaciones de Software Libre.

A continuación, se describen las actividades que contempla cada una de las fases de este proceso utilizando un flujograma de trabajo, en el que se indica la secuencia de ejecución de dichas actividades y una tabla en la que se describen las tareas que conforman cada una de ellas.

[2]Las iteraciones pueden llevarse a cabo en paralelo siempre y cuando lo permitan las dependencias entre las funcionalidades a construir.

1.3.1. Fase de Especificación de Requerimientos

En esta fase se especifican a detalle los requerimientos funcionales y no funcionales que debe cumplir el software para asegurar la calidad de éste, es decir, para satisfacer los requerimientos de los usuarios (IEEE STD 610-1990). Esta especificación se realiza para las funcionalidades que se construyen en cada iteración de desarrollo, de allí que el documento de especificación de requerimientos va evolucionando con cada iteración.

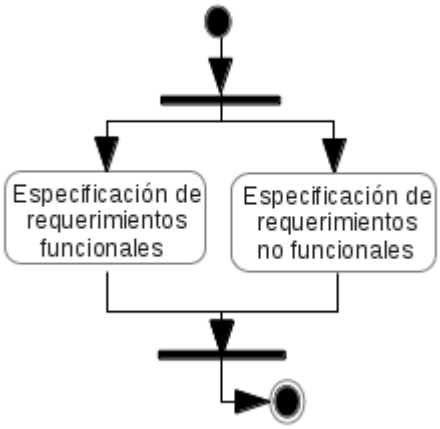


Figura 5 Flujo de trabajo de la fase de Especificación de Requerimientos

La especificación de requerimientos funcionales constituye la descripción de los casos de uso, en los cuales se indica la interacción entre los usuarios y las funcionalidades del software, es decir, las acciones que pueden ejecutar los usuarios en el software y las respuestas de éste ante dichas acciones. Es por ello que la especificación de requerimientos funcionales representa el principal insumo para la fase de Análisis y Diseño, así como para las fases de Codificación y Pruebas.

La especificación de requerimientos no funcionales corresponde a la definición de todas aquellas restricciones sobre el software que limitan la construcción del mismo. Entre estas restricciones se encuentran, por ejemplo, requerimientos referidos a: i) seguridad para el acceso a los datos que maneja un software; ii) intercambio de datos con otros software; iii) manejo de fallas en el software; iv) funcionamiento aceptable del software ante incrementos en el volumen de procesamiento.

En la Figura 5 se presenta el diagrama de flujo de trabajo concerniente a la secuencia de ejecución de las actividades contempladas para la fase de Especificación de Requerimientos. Luego, en la Tabla 3 se describen las tareas correspondientes a cada una de las actividades indicadas en el flujo respectivo.

Tabla 3: Tareas que integran las actividades asociadas a la fase de Especificación de Requerimientos.

Actividad: Especificación de requerimientos funcionales	
Tarea: Describir textualmente los casos de uso correspondientes a la iteración actual.	Recomendaciones: ▪ La descripción textual de un caso de uso se debe realizar en lenguaje natural, y la misma debe contener la siguiente información: 1. Actores: usuarios y/o sistemas que interactúan con el caso de uso. 2. Condiciones de entrada (precondiciones): condiciones que se deben cumplir para poder acceder al caso de uso. 3. Condiciones de salida (postcondiciones): respuesta del software al ejecutarse exitosamente el caso de uso. 4. Flujo básico: lista enumerada de los pasos que ejecuta el software y los actores para llevar a cabo el caso de uso, obteniéndose así la condición de salida. 5. Flujos alternativos: pasos que ejecutan los actores y el software en situaciones no comunes, como por ejemplo, los pasos referidos al ingreso de datos de entrada inválidos o la omisión de datos obligatorios, así como pasos referidos al comportamiento del software ante dicho ingreso u omisión de datos. 6. Requisitos especiales: cualquier otra información que se considere pertinente para la construcción de la función a la cual se hace referencia en el caso de uso. ▪ Los casos de uso no constituyen documentos de diseño de interfaz de usuario, por lo cual nunca debe hacerse referencia en ellos a elementos de la interfaz, tales como página principal, pantalla de ingreso o *clickear* botones [12]. Esta recomendación se sustenta en el hecho de que los casos de uso representan indicaciones de lo que se requiere construir, es decir, requerimientos funcionales, más no representan indicaciones de cómo estos requerimientos deben ser implementados, lo cual es una tarea del área de diseño de interfaz y codificación. Es importante destacar que cuando se incorporan elementos de interfaz en la descripción de los casos de uso, se tiende a generar largos y numerosos casos de uso que terminan siendo documentos poco atractivos para quienes deben implementar lo allí especificado. ▪ La descripción del caso de uso debe ser hecha en forma detallada, clara y precisa, de manera que el arquitecto de software, el diseñador gráfico, los programadores, los probadores y los documentadores no no tengan necesidad de leer ningún otro documento para realizar su trabajo en relación con el software a construir [13].
La tabla continúa en la siguiente página	

	▪ Para los requerimientos funcionales referidos a las operaciones crear, obtener, actualizar y borrar, que mantengan un comportamiento similar en el software y que sean cortas y simples, se recomienda especificar los mismos en un solo caso de uso. Este planteamiento evita especificar por separado cada una de las operaciones mencionadas, lo cual disminuye el número de casos de uso del software, y, por ende, el trabajo asociado a ello [14].
	Herramienta: ▪ Los diagramas de casos de uso y la descripción textual de los mismos pueden registrarse en el *wiki* correspondiente a "Especificación de requerimientos funcionales", incluido en el componente desarrollado para la plataforma *Trac*.
	Producto: ▪ Especificación de requerimientos funcionales. Este documento debe contener por cada requerimiento funcional: un diagrama de caso de uso (estos diagramas están contenidos en la propuesta de desarrollo del software) y su respectiva descripción textual.
	Responsables: ▪ Analistas.
	Colaboradores: ▪ Usuarios y/o interesados.
Tarea: Discutir con el equipo de desarrollo la descripción textual de los casos de uso asociados a la iteración actual.	Recomendaciones: ▪ Dado que la descripción textual de los requerimientos funcionales constituye el insumo principal que se utiliza para llevar a cabo las actividades de las fases de Análisis y Diseño, Codificación y Pruebas, se recomienda la discusión de esta descripción con los demás integrantes del equipo de desarrollo. Ello permite identificar posibles ambigüedades en dichas descripciones, así como agregar información de utilidad en éstas que no haya tomado en cuenta el Analista.

La tabla continúa en la siguiente página

	Productos: ▪ Minutas con los acuerdos establecidos sobre la descripción textual de los casos de uso. ▪ Especificación de requerimientos funcionales validada por el equipo de desarrollo (este documento incluye las mejoras a las que hubiera lugar en la especificación de requerimientos según la discusión realizada).
	Responsables: ▪ Analistas y demás miembros del equipo de desarrollo.
Tarea: Validar con los usuarios la descripción textual de los casos de uso asociados a la iteración actual.	Recomendaciones: ▪ Antes de pasar a la fase de Análisis y Diseño y/o a la fase de Codificación es muy importante que los usuarios validen la especificación de requerimientos, pues este documento refleja las tareas que los usuarios podrán realizar a través del software a desarrollar. De allí la importancia de considerar las observaciones de los usuarios respecto a dicho documento.
	Producto: ▪ Minutas con los acuerdos establecidos como resultado de la validación con los usuarios. ▪ Especificación de requerimientos funcionales validada por los usuarios (este documento incluye las mejoras a las que hubiera lugar según la validación realizada).
	Responsables: ▪ Analistas y usuarios.
Actividad: Especificación de requerimientos no funcionales	
Tarea: Definir con los usuarios los requerimientos no funcionales que debe cumplir el software.	Recomendaciones: ▪ Los requerimientos no funcionales representan un insumo muy importante para la definición de la arquitectura del software, debido a que implican aspectos que debe contemplar el software en términos de seguridad, tiempos de respuesta, interfaz de usuario, entre otros; que son determinantes para seleccionar una arquitectura que permita cumplir con tales requerimientos.
La tabla continúa en la siguiente página	

	▪ Durante el proceso de Conceptualización de Proyectos de Software Libre se levanta información que puede servir de insumo para la definición formal de requerimientos no funcionales. ▪ Estos requerimientos pueden ser definidos en la primera iteración y pueden ser refinados en iteraciones posteriores.
	Herramienta: ▪ La especificación de los requerimientos no funcionales puede registrarse en el wiki correspondiente a “Especificación de requerimientos no funcionales”, incluido en el componente desarrollado para la plataforma Trac.
	Producto: ▪ Especificación de requerimientos no funcionales.
	Responsables: ▪ Analistas y usuarios.

1.3.2. Fase de Análisis y Diseño

Esta fase comprende la definición de la arquitectura del software, la especificación de los datos persistentes y el diseño de la interfaz de usuario. En el caso de la arquitectura es conveniente destacar que ésta constituye un aspecto de fundamental importancia para el caso de aplicaciones de software que resulten complejas, ya sea en términos del número de requerimientos y/o el impacto de los mismos [15].

La arquitectura del software, la especificación de datos persistentes y el diseño de interfaz de usuario pueden construirse de forma incremental, en cada iteración de desarrollo, en base a los requerimientos funcionales y no funcionales que se aborden en cada iteración.

Las actividades y tareas que componen esta fase se describen a continuación.

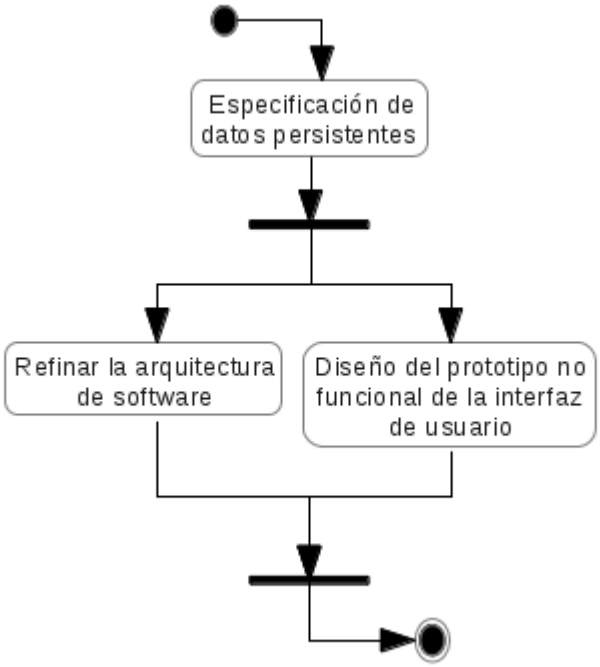


Figura 6 Flujo de trabajo de la fase de Análisis y Diseño

Tabla 4: Tareas que integran las actividades asociadas a la fase de Análisis y Diseño.

Actividad: Especificación de datos persistentes	
Tarea: Modelar los datos persistentes que maneja el software.	Recomendaciones: ▪ El modelado de datos representa el punto de partida en el diseño de una aplicación, pues en éste se identifican las entidades, sus atributos y las relaciones entre dicha entidades, lo cual nos permite definir con mayor claridad las funcionalidades que debe ofrecer el software en términos de los datos que maneja [16]. ▪ Para modelar los datos persistentes se debe seleccionar el tipo de modelo en el que se construirá el almacén de datos. Entre los tipos de modelos más utilizados se encuentran: orientado a objetos, entidad-relación, basado en documentos. Es posible realizar transformaciones entre estos modelos. ▪ De acuerdo a los requerimientos funcionales y no funcionales del software puede llegar a requerirse el uso de varios tipos de modelos de datos en un mismo desarrollo. ▪ Los objetos o entidades que maneja el software se pueden identificar en la descripción textual de los casos de uso. ▪ Los diagramas de clase elaborados como parte de la arquitectura del software constituyen en sí un modelo de datos, en caso de utilizar el modelo de datos orientado a objetos. ▪ En los casos en los que se trabaje con el modelo de datos relacional, se requiere modelar los datos del software en base al modelo entidad-relación. ▪ En la medida de lo posible, se recomienda utilizar tipos básicos para los campos de las relaciones, clases o documentos, de tal manera que la aplicación del software admita el uso de diversos gestores de datos que se ajusten a las necesidades del despliegue. ▪ Cuando se utilizan herramientas (editores gráficos) para modelar los datos es necesario tener en cuenta algunas pautas tales como, configurar la salida de los scripts en SQL estándar y mantener actualizado el modelo cuando ocurren cambios en los scripts de creación de la base de datos. ▪ Se debe tener como premisa que la aplicación podrá cambiar de gestor de datos, incluso de modelo de datos, por lo tanto, es conveniente ceñirse a estándares y patrones de base de datos, como por ejemplo, los definidos en SQL, en el Modelo-Vista-Controlador o en ORM (*Object-Relational Mapping*).
La tabla continúa en la siguiente página	

	Herramientas: ▪ Para elaborar diagramas de entidad-relación se pueden utilizar herramientas como: DIA, *ER Visual*, *BD Designer Fork* y *Druid*. ▪ El modelo de datos del software puede registrarse en el *wiki* de "Modelo de datos persistentes", incluido en el componente desarrollado para la plataforma *Trac*.
	Producto: ▪ Modelo de datos persistentes.
	Responsables: ▪ Analistas y/o el arquitecto de software.
Tarea: Especificar los datos a intercambiar con otras aplicaciones de software.	Recomendaciones: ▪ La interoperabilidad entre aplicaciones de software constituye un atributo de calidad a considerar en el desarrollo de aplicaciones que requieran el intercambio de datos con otras aplicaciones, razón por la cual es necesario especificar los datos a intercambiar, incluyendo en ello la definición de los formatos a utilizar para realizar dicho intercambio. ▪ Existen varias herramientas para facilitar la interoperabilidad entre aplicaciones de software, entre las más utilizadas se encuentran los Servicios Web [17].
	Herramienta: ▪ La especificación de datos a intercambiar puede registrarse en el *wiki* de "Modelo de datos persistentes", incluido en el componente desarrollado para la plataforma *Trac*.
	Producto: ▪ Especificación de datos a intercambiar.
	Responsables: ▪ Analistas y/o arquitecto de software.

La tabla continúa en la siguiente página

<table>
<tr><td></td><td>Colaboradores:
Equipo de desarrollo.</td></tr>
<tr><td colspan="2">Actividad: Refinar la arquitectura de software</td></tr>
<tr><td rowspan="3">Tarea: Refinar la arquitectura de software.</td><td>Recomendaciones:
En vista de que las fases de Especificación de Requerimientos y Análisis y Diseño brindan, en cada iteración de desarrollo, una mayor comprensión sobre el software, se puede refinar la arquitectura planteada en la conceptualización del proyecto en cada iteración. El refinar la arquitectura incluye una descripción detallada de la misma, la cual puede ser representada a través de diferentes diagramas o vistas arquitectónicas, que constituyen las diferentes perspectivas del diseño de una aplicación [18]. Entre los diagramas utilizados para representar vistas arquitectónicas se encuentran: diagramas de clases, diagramas de secuencia, diagramas de estado, diagramas de componentes, entre otros [19]. Cabe destacar que no es obligatorio diseñar todas las vistas arquitectónicas de un software, basta con plantear aquellas que se consideren pertinentes según la complejidad y alcance del software, que permitan contar con una visión de éste que sirva de apoyo en las diferentes fases de su construcción, así como en los procesos de mantenimiento.
La descripción textual de los casos de uso de cada iteración, constituye el insumo base para identificar y/o refinar componentes de la arquitectura de software.</td></tr>
<tr><td>Herramientas:
Existen varias herramientas para elaborar los diagramas de la arquitectura de un software, entre ellas: Dia, Umbrello, Bonita, CASEUML, ArgoUML, BOUML, entre otras.
En el caso de CENDITEL, se propone utilizar para elaborar los diagramas de arquitectura de un software la herramienta Platuml, contenida en el componente desarrollado para la plataforma Trac. Entre los diagramas que se pueden elaborar con esta herramienta se encuentran: diagramas de clases, de secuencia, de estado, entre otros.
Los diagramas en base a los cuales se represente la arquitectura del software pueden registrarse en el wiki de “Arquitectura del software”, incluido en el componente desarrollado para la plataforma Trac.</td></tr>
<tr><td>Productos:
Diagramas que representan la arquitectura del software.</td></tr>
<tr><td colspan="2">La tabla continúa en la siguiente página</td></tr>
</table>

<table>
<tr><td></td><td>Responsable:
Arquitecto de software.</td></tr>
<tr><td></td><td>Colaboradores:
Analistas, otros integrantes del equipo de desarrollo.</td></tr>
<tr><td colspan="2">Actividad: Diseño del prototipo no funcional de la interfaz de usuario</td></tr>
<tr><td rowspan="4">Tarea: Discutir con el equipo de desarrollo el diseño de la interfaz más adecuada para el software a desarrollar.</td><td>Recomendaciones:
Para facilitar la actividad de diseño de interfaz y asegurar un diseño adecuado según las limitaciones y/o restricciones que se puedan presentar, tanto a nivel de construcción como a nivel de usuario, se sugiere realizar reuniones entre los programadores, los analistas y los diseñadores gráficos para discutir el diseño de interfaz más adecuado para el software a desarrollar.
A fin de fundamentar de manera explicita las decisiones de diseño gráfico, así como brindar apoyo al proceso de construcción de la interfaz del software, se sugiere elaborar un documento en el que se contemplen las decisiones de diseño gráfico que se discutan en el equipo de desarrollo.</td></tr>
<tr><td>Producto:
Documento de fundamentación del diseño de interfaz del software.</td></tr>
<tr><td>Responsables:
Diseñador gráfico y demás integrantes del equipo de desarrollo.</td></tr>
<tr><td>Colaboradores:
Usuarios.</td></tr>
<tr><td colspan="2">La tabla continúa en la siguiente página</td></tr>
</table>

<table>
<tr>
<td>Tarea: Diseñar las pantallas no funcionales correspondientes a la interfaz gráfica del software.</td>
<td>Recomendaciones:
Por lo general toda interfaz gráfica se compone de los siguientes elementos:
Encabezado: se refiere a un logo o imagen de identificación del software. Se recomienda utilizar frames u otras herramientas que permitan que el encabezado se cargue sólo una vez en las pantallas de interfaz del software.
Menú: muestra el listado de las funcionalidades que ofrece el software. Se recomienda que el menú se ubique, de ser posible, en varios lugares de las pantallas de interfaz.
Zona de contenido: muestra las operaciones que puede ejecutar el usuario conformes a las funcionalidades que ofrece el software.
Zona de mensajes: muestra los mensajes de tipo informativo, de error y de éxito en una operación del software [20].
Se recomienda utilizar un diagrama de navegabilidad entre las pantallas diseñadas para representar la interfaz de usuario, con lo cual se podrá mostrar el paso de una pantalla a otra.
Teniendo en cuenta la importancia que adquiere la interfaz de usuario, en términos de usabilidad, se plantean a continuación algunas recomendaciones a considerar para el diseño de la interfaz gráfica:
La interfaz de las operaciones que ejecuta el software debe mantener un estándar visual. Por ejemplo, de ser posible, las funciones para modificar o eliminar información deben seguir un mismo esquema de presentación para las distintas funciones u operaciones del software en las cuales éstas se requieran.
La estructura de iconos o botones que sirven de enlace a las funciones que ejecuta el software debe ser lo más sencilla posible, es decir, se debe evitar la ejecución de varios pasos a efectuar en el software para acceder a alguna de sus funciones.
La interfaz debe mantener una estandarización en relación al formato de los iconos o botones mostrados.
Los botones o iconos utilizados en la interfaz pueden mostrar textos en los cuales se indique la función que cumple cada uno de éstos. Para ello se recomienda hacer uso de los epígrafes o etiquetas.
Los tipos y tamaños de las letras utilizadas en las pantallas deben facilitar la visualización de los textos o frases que se presentan en la interfaz.
Los colores utilizados en cada pantalla deben ser contrastantes entre sí, a fin de facilitar la lectura de la información mostrada en la interfaz.
La interfaz debe mantener una estandarización en relación al idioma de las personas que usarán el software.</td>
</tr>
<tr>
<td colspan="2" align="center">La tabla continúa en la siguiente página</td>
</tr>
</table>

	8. Se debe considerar en el diseño de la interfaz lineamientos de usabilidad asociados a la plataforma en la que se desarrolle el software, así como lineamientos de usabilidad para discapacitados. Por ejemplo, las plataformas de desarrollo como *Android*, *KDE*, *Web*, entre otras, cuentan con una serie de lineamientos de usabilidad para ser aplicados en los software que se desarrollen en éstas. ▪ Además de los lineamientos de usabilidad mencionados, existen pautas así como estándares que proporcionan apoyo importante en el diseño de la interfaz de usuario, entre ellas podemos citar algunas referencias como: *Stone et al*. (2005), *Tidwell* (2011), *Luzardo* (2009); *ISO 9421*; *ISO 14915*; *ISO 13407*; *ISO / CD 20282*.
	Herramientas: ▪ La herramienta *Pencil* se puede utilizar para elaborar las pantallas del prototipo no funcional de la interfaz de usuario. ▪ Este prototipo puede registrarse en el *wiki* “Prototipo no funcional de la interfaz de usuario”, incluido en el componente desarrollado para la plataforma *Trac*.
	Producto: ▪ Prototipo no funcional de la interfaz de usuario.
	Responsable: ▪ Diseñador Gráfico.
	Colaboradores: ▪ Equipo de desarrollo, usuarios.
Tarea: Validar el prototipo no funcional de la interfaz con los usuarios.	Recomendaciones: ▪ La interfaz constituye uno de los elementos mas importantes del software en términos del usuario, pues ella representa el puente de interacción entre el software y éste. De allí, la relevancia significativa de considerar a los usuarios como parte importante del equipo de diseño de la interfaz.

La tabla continúa en la siguiente página

	Productos: ▪ Minutas con los acuerdos establecidos como resultado de la validación con los usuarios. ▪ Prototipo no funcional de la interfaz validado por los usuarios (este documento incluye las mejoras a las que hubiera lugar según la validación realizada).
	Responsables: ▪ Diseñador gráfico y usuarios.
	Colaboradores: ▪ Demás miembros del equipo de desarrollo.

1.3.3. Fase de Codificación

En esta fase se codifican las funcionalidades de la aplicación de software correspondientes a la iteración actual, se construye la interfaz de usuario y la base de datos. De esta manera, con cada iteración de desarrollo se obtiene una nueva versión de la aplicación clasificada como versión beta; es decir, una versión sobre la cual se deben realizar un conjunto de pruebas funcionales y no funcionales. Las actividades y tareas que componen esta fase se describen a continuación.

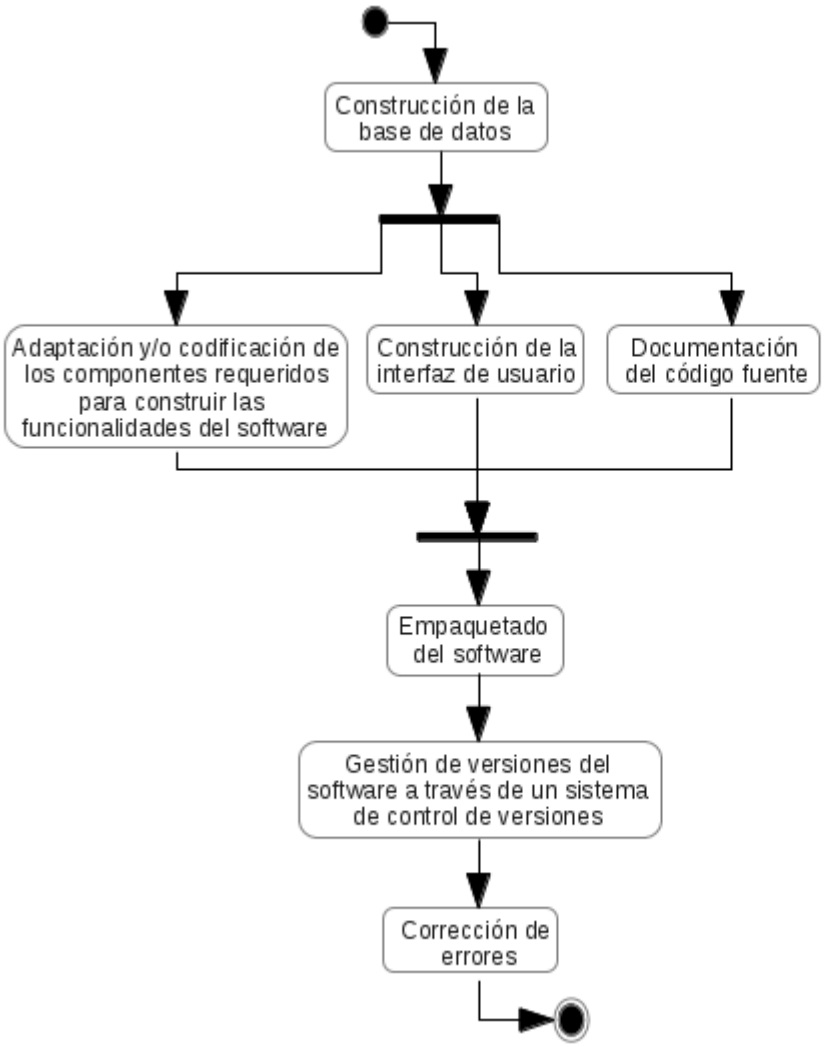


Figura 7 Flujo de trabajo de la fase de Codificación

Tabla 5: Tareas que integran las actividades asociadas a la fase de Codificación.

Actividad: Construcción de la base de datos	
Tarea: Construir la base de datos conforme al modelo de datos persistentes.	Recomendaciones: ▪ Es importante mencionar que los esquemas de base de datos que generan automáticamente algunas herramientas para modelado de datos, así como para implementación de arquitectura de software, entre ellas el entorno de desarrollo (*framework*) *Django*, pueden no manejar de forma satisfactoria todas las partes de estos esquemas, razón por la cual puede requerirse la programación directa de estas partes. ▪ Otro asunto importante a tomar en cuenta es el referido a la inicialización de la base de datos, pues ello se puede facilitar a través de la carga automática de datos.
	Herramientas: ▪ Algunas herramientas que permiten generar *scripts* para base de datos conforme al modelo de datos: *ER Visual*, *BD Designer Fork*. ▪ Mapeo Objeto-Relacional (ORM).
	Producto: ▪ Base de datos.
	Responsables: ▪ Programadores.
	Colaboradores: ▪ Equipo de desarrollo.
Actividad: Adaptación y/o codificación de los componentes requeridos para construir las funcionalidades del software	
Tarea: Codificar los componentes requeridos para construir las funcionalidades asociadas a la iteración actual.	Recomendaciones: ▪ Para agilizar la fase de codificación y mejorar la calidad del software se recomienda el uso de entornos de desarrollo (*framework*), así como de patrones de programación y componentes reutilizables que puedan ser adaptados según se requiera.
La tabla continúa en la siguiente página	

<table>
<tr><td></td><td>Entre los patrones de programación que se pueden utilizar se encuentran: patrón de acumulación, patrón lectura de datos, patrón de conteo [21], patrón Super Lazo (Super Loop), patrón Fondo/Portada (Background/Foreground) [22], entre otros.</td></tr>
<tr><td></td><td>Producto:
Componentes implementados, componentes adaptados.</td></tr>
<tr><td></td><td>Responsables:
Programadores.</td></tr>
<tr><td>Tarea: Realizar las pruebas unitarias a los componentes adaptados y/o codificados y corregir los errores encontrados.</td><td>Recomendaciones:
A fin de evitar errores al momento de integrar los componentes del software se recomienda elaborar pruebas unitarias para verificar el funcionamiento, por separado, de cada uno de estos componentes. Para ello se recomienda utilizar herramientas que generen automáticamente datos de entrada a los componentes y que comprueben las salidas que éstos generan.
La utilización de herramientas automatizadas para pruebas unitarias juega un papel fundamental en la construcción de software, dada la cantidad de componentes que se pueden construir y/o adaptar para un software determinado y la complejidad de éstos.</td></tr>
<tr><td></td><td>Herramientas:
Para aplicar pruebas unitarias se pueden utilizar herramientas como: PhpUnit (para lenguaje PHP), xUnit (para distintos lenguajes de programación), entre otras.</td></tr>
<tr><td></td><td>Producto:
Errores corregidos.</td></tr>
<tr><td></td><td>Responsables:
Programadores.</td></tr>
<tr><td colspan="2">La tabla continúa en la siguiente página</td></tr>
</table>

<table>
<tr><th colspan="2">Actividad: Construcción de la interfaz de usuario</th></tr>
<tr><td rowspan="3">Tarea: Construir la interfaz de usuario correspondiente a las funcionalidades asociadas a la iteración actual.</td><td>Recomendaciones:
Para crear interfaces de usuario se pueden utilizar herramientas como:
TkInter, wxPython, PyGTK, PyQt, estas herramientas permiten crear interfaces gráficas en Python.
El entorno de desarrollo Qt permite crear interfaces gráficas tanto para aplicaciones de escritorio como para móviles.
Codeblocks, CodeLite, Gtkmm, entre otras.
La herramienta Cascading Style Sheets (CSS) es comúnmente utilizada para la creación de hojas de estilos de interfaz gráfica.</td></tr>
<tr><td>Producto:
Interfaz gráfica del software.</td></tr>
<tr><td>Responsables:
Diseñador gráfico, programadores.</td></tr>
<tr><th colspan="2">Actividad: Documentación del código fuente</th></tr>
<tr><td rowspan="2">Tarea: Documentar el código fuente.</td><td>Recomendaciones:
La documentación del código fuente representa una actividad fundamental para facilitar los procesos de apropiación del software (mantenibilidad), por lo cual debe ser considerada como una práctica cotidiana durante la construcción del código. En este sentido, se recomienda realizar la documentación de todos los componentes del software, tanto los codificados como los reutilizados.
Se sugiere que la documentación de cada componente (función, método, clase), como mínimo, haga referencia a qué hace el componente, qué parámetros hay que pasarle y qué devuelve [23]. El funcionamiento interno de los componentes debe poder comprenderse al revisar el código fuente, y en los casos de componentes complejos se sugiere incluir comentarios que expliquen con mayor detalle el funcionamiento de los mismos.</td></tr>
<tr><td>Herramientas:
Para generar la documentación del código fuente se pueden utilizar herramientas como: Doxygen, phpDocumentor, entre otras.</td></tr>
<tr><td colspan="2">La tabla continúa en la siguiente página</td></tr>
</table>

<table>
<tr><td></td><td>Producto:
Código documentado.</td></tr>
<tr><td></td><td>Responsable:
Programadores.</td></tr>
<tr><td colspan="2">Actividad: Empaquetado del software</td></tr>
<tr><td>Tarea: Empaquetar cada versión estable del software para diferentes distribuciones.</td><td>Recomendaciones:
A fin de que el software desarrollado pueda ser portable, es decir, operar en varias distribuciones, se recomienda empaquetar el software para más de una distribución.
Para facilitar y agilizar las actividades que involucra el empaquetado del software se sugiere configurar un ambiente de máquinas virtuales para distribuciones específicas de software, en las que se encuentren automatizadas las labores de empaquetado requeridas en cada una de estas distribuciones.
El empaquetar el software facilita en gran medida el proceso de su instalación.
Conforme a la complejidad del desarrollo y/o a la diferencia entre las versiones del software, se debe dejar a decisión del equipo de desarrollo si el empaquetado se realiza al culminar el desarrollo del software o si se realiza para versiones intermedias de éste.</td></tr>
<tr><td></td><td>Productos:
Versiones del software empaquetadas para varias distribuciones.</td></tr>
<tr><td></td><td>Responsable:
Programadores.</td></tr>
<tr><td colspan="2">Actividad: Gestión de las versiones del software a través de un Sistema de Control de Versiones (SCV)</td></tr>
<tr><td>Tarea: Colocar el código fuente desarrollado en cada iteración en el SCV que se utilice en el proyecto.</td><td>Recomendaciones:
El uso de SCV es muy importante para el manejo organizado de las distintas versiones que se desarrollan para un software, pues éstos permiten llevar un control sobre los cambios que se realizan sobre estas versiones, facilitando así el trabajo colaborativo en torno al desarrollo de software.</td></tr>
<tr><td colspan="2">La tabla continúa en la siguiente página</td></tr>
</table>

<table>
<tr><td></td><td>

Es importante resaltar que el colocar el código fuente en SCV debe ser una tarea cotidiana para la práctica de desarrollo, dado que dicha tarea representa un elemento clave para la conformación de una práctica orientada a la colaboración. En este sentido, cabe destacar que el sistema de control de versiones debe contener tanto versiones beta como versiones alfa del software.
Para mayor organización del código en el SCV, así como para facilitar el trabajo colaborativo en torno a las versiones del software, se sugiere que cada envío (commit) de código al SVC comprenda cambios en torno a un solo asunto, y esté acompañado de una descripción suficiente y apropiada. Esta forma de organización del código es una tarea establecida en proyectos notables, como por ejemplo, el desarrollo del núcleo Linux.

</td></tr>
<tr><td></td><td>Herramientas:

Existen muchas herramientas para el control de versiones de software, entre ellas tenemos: CVS, SVK, Subversion, SourceSafe, Mercurial, Bazaar, GIT, Darcs.
La mayoría de las plataformas para gestión de proyectos de software contienen sistemas para el control de versiones, tal es el caso de la plataforma Trac que se utiliza en CENDITEL.

</td></tr>
<tr><td></td><td>Producto:

Código fuente del software en el sistema de control de versiones.

</td></tr>
<tr><td></td><td>Responsable:

Programadores.

</td></tr>
<tr><td colspan="2">Actividad: Corrección de errores</td></tr>
<tr><td>Tarea: Corregir los errores reportados.</td><td>Recomendaciones:

Al utilizar un sistema de gestión de incidentes cada Programador puede tener acceso al reporte de los errores que se le asignen para su respectiva corrección.
Una vez corregido el error se recomienda al Programador registrar en el sistema de gestión de incidentes la causa del error y la corrección realizada. Esto permite llevar un historial sobre la resolución de errores, el cual puede ser de utilidad para solventar en el futuro errores iguales o similares que se puedan presentar.

</td></tr>
<tr><td colspan="2">La tabla continúa en la siguiente página</td></tr>
</table>

	Productos: ▪ Código corregido. ▪ Registro de correcciones en el sistema de control de errores.
	Responsable: ▪ Programadores.
	Colaboradores: ▪ Usuarios.

1.3.4. Fase de Pruebas

En esta fase se elaboran y aplican pruebas funcionales y no funcionales a cada versión del software, así como pruebas de regresión y de instalación/desinstalación, con lo cual se facilita la detección temprana de errores y/o incompatibilidades en el código. Las pruebas de instalación/desinstalación son de fundamental importancia para la apropiación del software, pues los usuarios deben poder instalar éste para comenzar el proceso de pruebas, apoyando así el desarrollo y mejora del software.

Las pruebas mencionadas deben ser elaboradas y aplicadas por probadores de software, quienes, se recomienda, deben ser personas distintas a quienes codifican la aplicación, dado que los programadores fijan mayor interés en lo que se supone que debe hacer el código que en lo que éste hace. De allí que los probadores suelen detectar errores no percibidos por lo programadores [24].

Las actividades y tareas que componen la fase de Pruebas se describen a continuación.

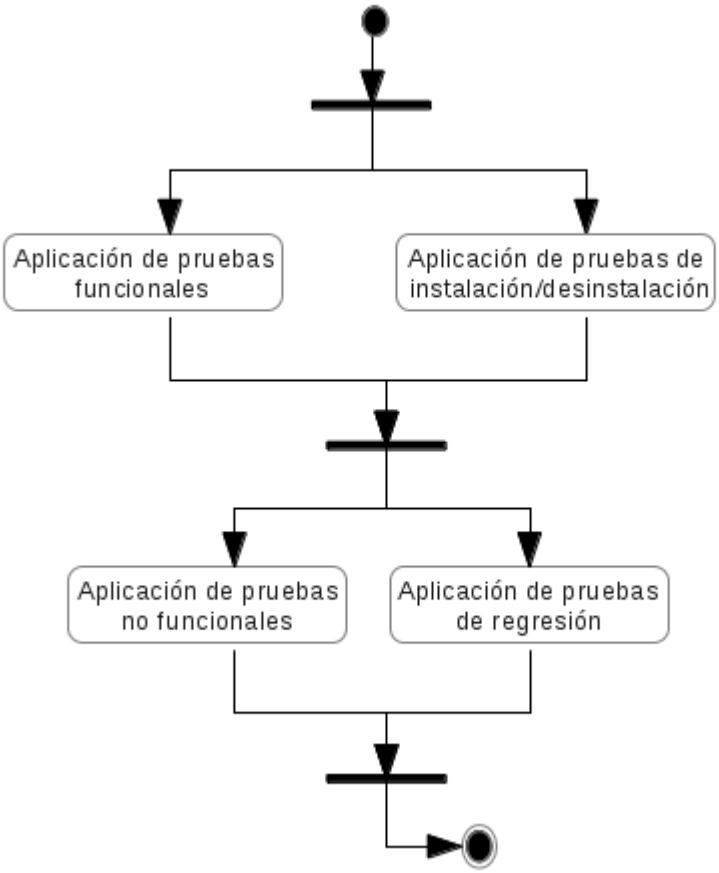


Figura 8 Flujo de trabajo de la fase de Pruebas

Tabla 6: Tareas que integran las actividades asociadas a la fase de Pruebas.

Actividad: Aplicación de pruebas funcionales	
Tarea: Elaborar el plan de pruebas funcionales correspondientes a las funcionalidades desarrolladas en la iteración actual.	Recomendaciones: ▪ Las pruebas funcionales se utilizan para verificar que el software ejecute sus funciones según lo establecido en la especificación de requerimientos funcionales. Por esta razón, la elaboración de las pruebas funcionales se basa en la descripción textual de los casos de uso definidos para el software. ▪ Un plan de pruebas funcionales se compone de casos de prueba con los cuales se verifica el comportamiento de las funcionalidades del software, en términos de los escenarios indicados en la descripción textual de los casos de uso asociados a dichas funcionalidades, es decir, en términos de flujos básicos, flujos alternativos y requisitos especiales. ▪ A continuación se presenta un formato o plantilla para la descripción de casos de prueba basado en la técnica de pruebas Caja Negra (*black-box testing*) [25], en éste se indica el tipo de información que debe contener cada caso de prueba: 1. Escenario bajo el cual se realiza el caso de prueba, es decir, debe indicar si la prueba se realiza para el flujo básico, para actividades específicas dentro del flujo básico, para flujos alternativos o para requisitos especiales del caso de uso respectivo. 2. Objetivo del caso de prueba, en donde se indica el propósito que se persigue al ingresar u omitir datos de entrada al software para un escenario específico. 3. Pre-condición, es un requisito que debe cumplir el software para poder ejecutar el caso de prueba. 4. Pasos para realizar el caso de prueba, donde se indican los pasos que debe llevar a cabo el probador en el software para ejecutar la prueba, así como los datos que debe registrar. Para ilustrar lo expuesto se presenta el siguiente ejemplo: • Entrar a la función “Registrar proyecto”. • Ingresar los siguientes datos: a) Título del proyecto: omitir titulo. b) Fecha de inicio del proyecto: 12/06/2014. c) Fecha de culminación del proyecto: 25/12/2015. • Pulsar la opción “Guardar”.
La tabla continúa en la siguiente página	

<table>
<tr><td></td><td>5. Salida esperada, es decir, el resultado que debe emitir el software, según el escenario de prueba.
6. Salida obtenida, acá se indica si el comportamiento del software al aplicar el caso de prueba respectivo es igual o diferente a la salida esperada. Una salida obtenida que sea diferente a la salida esperada se considera un error o una falla en el software.
▪ Los datos utilizados en los casos de prueba deben ser de tipo válidos e inválidos, a fin de verificar el comportamiento del software ante estos tipos de datos. Por lo general, el comportamiento del software ante datos de entrada inválidos es especificado en los flujos alternativos de los casos de uso definidos para el software.
▪ Para agilizar la elaboración y aplicación de las pruebas funcionales se sugiere, de ser posible, abarcar varios escenarios en un mismo caso de prueba [26]. Por ejemplo, se podrían probar varios flujos alternativos en un mismo caso de prueba.</td></tr>
<tr><td></td><td>Herramientas:
▪ Existen varias técnicas de prueba, para el caso de esta metodología se utiliza la técnica de diseño de pruebas Caja Negra.
▪ El plan de pruebas puede registrarse en el wiki “Plan de pruebas funcionales”, incluido en el componente desarrollado para la plataforma Trac.</td></tr>
<tr><td></td><td>Producto:
▪ Plan de pruebas funcionales.</td></tr>
<tr><td></td><td>Responsables:
▪ Probadores.</td></tr>
<tr><td>Tarea: Aplicar el plan de pruebas funcionales correspondiente a las funcionalidades desarrolladas en la iteración actual.</td><td>Recomendaciones:
▪ En vista de que las pruebas funcionales pueden utilizarse como pruebas de regresión, se recomienda utilizar herramientas que permitan grabar la ejecución de las pruebas funcionales, de modo que éstas puedan ser reproducidas posteriormente para verificar el comportamiento del software después de haber modificado funcionalidades o haber corregido errores.</td></tr>
<tr><td colspan="2">La tabla continúa en la siguiente página</td></tr>
</table>

<table>
<tr><td></td><td>

- Los errores que se encuentren al aplicar las pruebas funcionales deben reportarse al equipo de desarrollo, a fin de que éstos gestionen la corrección de los mismos. Para reportar un error se debe indicar el caso de prueba o funcionalidad en la cual ocurre el error, así como una captura de la pantalla donde se indica el error en el software.

</td></tr>
<tr><td></td><td>

Herramientas:

- Existen diversas herramientas para aplicar pruebas funcionales, entre estas se encuentran: *Selenium* (permiten grabar las pruebas), *Jmeter*, *Watir* (para pruebas de aplicaciones web en *Ruby*), *SOLEX* (permiten grabar las pruebas), entre otras.
- El reporte de errores se puede hacer por vía correo electrónico, a través de mecanismos de reporte o directamente a través de un sistema para gestión de errores.

</td></tr>
<tr><td></td><td>

Producto:

- Reporte de pruebas funcionales.

</td></tr>
<tr><td></td><td>

Responsables:

- Probadores.

</td></tr>
<tr><td colspan="2">Actividad: Aplicación de pruebas de instalación/desinstalación</td></tr>
<tr><td>Tarea: Probar el proceso de instalación/ desinstalación de la aplicación de software en los hardware y software en que ésta pueda operar.</td><td>

Recomendaciones:

- Las pruebas de instalación/desinstalación básicamente radican en ejecutar los pasos definidos para tales acciones en el manual de usuarios. Con estas pruebas se busca verificar que el proceso de instalación/desinstalación de la aplicación, no presente fallas.
- Estas pruebas también pueden automatizarse en algunos casos. Por ejemplo, *Debian* tiene el sistema "*Debian Continuous Integration*" (http://ci.debian.net/) basado en *autopkgtest*.
- Los errores que se encuentren en el proceso de instalación/desinstalación deben reportarse al equipo de desarrollo, a fin de que éste gestione la corrección de los mismos. Para reportar un error se debe presentar una captura de la pantalla donde se indica el error en el software durante el proceso de instalación/desinstalación.

</td></tr>
<tr><td colspan="2">La tabla continúa en la siguiente página</td></tr>
</table>

<table>
<tr><td></td><td>

- Posterior a la corrección de los errores que se reporten durante este tipo de pruebas, éstas se deben repetir para verificar que el proceso de instalación/desinstalación se lleve a cabo sin presentar fallas.

</td></tr>
<tr><td></td><td>

Herramientas:

- Para reportar las fallas del proceso de instalación/desinstalación se pueden utilizar las herramientas planteadas para el reporte de errores en pruebas funcionales.
- Para pruebas automatizadas de compilación y construcción de paquetes puede usarse herramientas como *buildd* o *pbuilder*.

</td></tr>
<tr><td></td><td>

Producto:

- Reporte de pruebas de instalación/desinstalación.

</td></tr>
<tr><td></td><td>

Responsables:

- Probadores.

</td></tr>
<tr><td colspan="2">Actividad: Aplicación de pruebas no funcionales</td></tr>
<tr><td>Tarea: Elaborar el plan de pruebas no funcionales correspondientes a las funcionalidades desarrolladas en la iteración actual.</td><td>

Recomendaciones:

- Las pruebas no funcionales se utilizan para verificar en el software el cumplimiento de los atributos de calidad (requerimientos no-funcionales) establecidos para el mismo. Existen varios tipos de pruebas no funcionales, entre éstas se encuentran: pruebas de seguridad, pruebas de rendimiento, pruebas de usabilidad y pruebas de portabilidad. Entre las pruebas no funcionales más aplicadas se encuentran las pruebas de rendimiento [27], con las cuales se estudia el comportamiento del software ante diversos niveles de carga de entrada de datos, de actividades o de almacenamiento.
- Cabe destacar la prioridad que adquiere la seguridad en términos de la calidad de las aplicaciones de software, de allí la importancia de las pruebas de vulnerabilidad[3].

</td></tr>
<tr><td colspan="2">La tabla continúa en la siguiente página</td></tr>
</table>

[3]En las páginas web que se muestran a continuación se detalla información de interés sobre el tema de pruebas de seguridad o vulnerabilidad:

- http://www.vencert.gob.ve/index.php/vencert/biblioteca
- http://www.isecom.org/

	Herramienta: ▪ El plan de pruebas puede registrarse en el *wiki* “Plan de pruebas no funcionales”, incluido en el componente desarrollado para la plataforma *Trac*.
	Producto: ▪ Plan de pruebas no funcionales.
	Responsables: ▪ Probadores.
Tarea: Aplicar el plan de pruebas no funcionales correspondientes a las funcionalidades desarrolladas en la iteración actual.	Recomendaciones: ▪ La aplicación de la mayoría de las pruebas de rendimiento de forma manual es poco factible, dado que se requiere de una gran cantidad de probadores utilizando el software en un mismo período de tiempo, por lo cual se recomienda el uso de herramientas de prueba y/o la automatización de éstas. ▪ Los resultados de las pruebas de rendimiento deben indicar cuando el software falla para determinados valores de las variables que se estudian en estas pruebas. Por ejemplo, en una prueba de rendimiento en la cual se estudia la variable “número de usuarios conectados al mismo tiempo”, se debe poder indicar como resultado de la prueba el número de usuarios que soporta el software conectados al mismo tiempo antes de presentar fallas de conectividad o de operación. ▪ Los responsables de aplicar las pruebas de rendimiento deben contar con conocimientos y habilidades en el área de programación, dado que estas pruebas, por lo general, implican la simulación de un número de usuarios interactuando con las funcionalidades del software. ▪ Los errores del software que se encuentren al aplicar las pruebas no funcionales deben reportarse al equipo de desarrollo, a fin de que éste gestione la corrección de los mismos.
	Herramientas: ▪ Para aplicar pruebas de rendimiento se pueden utilizar herramientas como: *JCrawler*, *SOLEX* (también se puede usar para aplicar pruebas de regresión), *Multi-mechanize*, entre otras. ▪ Para aplicar pruebas de seguridad se pueden utilizar herramientas como: *Powerfuzzer*, *Nessus*, *Netcat*, *John the Ripper* y *OpenSSH*. ▪ Para reportar los resultados de las pruebas no funcionales se pueden utilizar las herramientas planteadas para el reporte de errores en pruebas funcionales.
La tabla continúa en la siguiente página	

<table>
<tr><td></td><td>Producto:
Reporte de pruebas no funcionales.</td></tr>
<tr><td></td><td>Responsables:
Probadores.</td></tr>
<tr><td></td><td>Colaboradores:
Programadores.</td></tr>
<tr><td colspan="2">Actividad: Aplicación de pruebas de regresión</td></tr>
<tr><td rowspan="5">Tarea: Aplicar pruebas de regresión a la versión del software obtenida en la iteración actual.</td><td>Recomendaciones:
Las pruebas de regresión se utilizan, por lo general, después de la corrección de errores o modificaciones en el código que puedan generar alteraciones considerables en el software, como por ejemplo, introducción de errores. Las pruebas de regresión se definen en sí como pruebas funcionales [28], de allí la recomendación de utilizar herramientas que permitan grabar las pruebas funcionales aplicadas al software, con el fin de poder reproducirlas como pruebas de regresión.
Existen tres tipos de pruebas de regresión: pruebas para todas las funcionalidades del software, pruebas sólo para las funcionalidades que han sido modificadas y pruebas para las funcionalidades o piezas de código que se puedan ver afectadas por las modificaciones realizadas en el software.
Para reportar los errores encontrados en las pruebas de regresión, se sugiere utilizar el mismo procedimiento indicado para el caso de reporte de errores en pruebas funcionales.</td></tr>
<tr><td>Herramientas:
Para reportar los errores encontrados en las pruebas de regresión, se pueden utilizar las herramientas planteadas para el reporte de errores en pruebas funcionales.</td></tr>
<tr><td>Producto:
Reporte de pruebas de regresión.</td></tr>
<tr><td>Responsables:
Probadores.</td></tr>
</table>

1.3.5. Fase de Liberación

En esta fase se llevan a cabo una serie de actividades asociadas a la liberación de versiones del software. Entre estas actividades se encuentran: elaboración de manuales, empaquetado del software, publicación de versiones beta (versiones de prueba) y versiones estables.

Las actividades y tareas que componen esta fase se describen a continuación.

Figura 9 Flujo de trabajo de la fase de Liberación

Tabla 7: Tareas que integran las actividades asociadas a la fase de Liberación.

Actividad: Elaboración del manual de usuarios	
Tarea: Elaborar el manual de usuarios.	Recomendaciones: ▪ A fin de que el manual de usuarios sea una herramienta de apoyo para el uso del software, se sugiere que éste contenga la siguiente información: 1. Índice. 2. Descripción general del software y de sus funciones, así como de las tareas que pueden ser ejecutadas utilizando el software. 3. Descripción de los símbolos y convenciones presentadas en la interfaz. 4. Explicación del proceso de instalación/desinstalación del software. En este caso se sugiere indicar la siguiente información: *a*) Tipo de procesador, memoria RAM y dispositivos de entrada y salida de datos (*Cd-Rom*, micrófono, teclado, escáner, CD y/o DVD, altavoces, auriculares, tarjeta de sonido, impresoras, etc.) requeridos para colocar el software en funcionamiento, así como el tamaño (capacidad) del dispositivo de almacenamiento que se requiere. *b*) Sistemas operativos en los cuales funciona el software. *c*) Paquetes de software requeridos para la instalación en los sistemas operativos indicados. *d*) Información respectiva a la configuración de software en los respectivos sistemas operativos. *e*) Pasos para instalar y desinstalar el software. En caso de existir un procedimiento automatizado para instalar y desintalar éste debe ser indicado. 5. Explicación sobre el modo de uso de cada una de las funciones que componen el software, incluyendo capturas de pantalla que el software presenta para cada una de estas funciones. 6. Preguntas frecuentes, donde se indican algunas de las preguntas más comunes que podría realizar el usuario con respecto al uso del software. 7. Ejemplos para ayudar en la compresión de asuntos determinados. 8. Explicación de los mensajes de error, cuando se considere necesario.
La tabla continúa en la siguiente página	

	Herramientas: ▪ Para elaborar manuales de usuarios se pueden utilizar herramientas como *Sphinx*. ▪ El manual puede registrarse en el *wiki* “Manual de Usuarios”, incluido en el componente desarrollado para la plataforma *Trac*.
	Producto: ▪ Manual de usuarios.
	Responsable: ▪ Documentador.
Tarea: Verificar correspondencia del manual de usuarios con el software desarrollado.	Recomendaciones: ▪ La revisión del manual de usuarios debe realizarse a fin de verificar que éste se encuentre en correspondencia con el software desarrollado, así como para verificar que la información presentada en el manual sea clara, precisa y de fácil comprensión para los usuarios.
	Producto: ▪ Observaciones sobre las revisiones realizadas al manual de usuarios.
	Responsables: ▪ Analistas, probadores o cualquier miembro del equipo de desarrollo.
Publicación de versiones del software	
Tarea: Publicar la versión beta del software correspondiente a la iteración actual.	Recomendaciones: ▪ La publicación de versiones beta es de fundamental importancia para el proceso de aseguramiento de calidad de la aplicación que se desarrolla, puesto que esta publicación se realiza con la finalidad de aumentar el número de personas que prueben el software, lo cual incrementa la posibilidad de encontrar errores, permitiendo así mejorar la calidad de la aplicación que se desarrolla. ▪ Con la publicación de versiones beta no sólo se busca que los usuarios prueben la aplicación a fin de reportar errores, sino también que éstos puedan plantear observaciones respecto a la interfaz de la aplicación y/o a las funcionalidades de la misma.
La tabla continúa en la siguiente página	

	Producto: ▪ Versión beta publicada.
	Responsables: ▪ Líder del proyecto y equipo de desarrollo.
Tarea: Publicar la versión estable del software correspondiente a la iteración actual.	Recomendaciones: ▪ Las versiones estables se publican una vez se hayan corregido los errores reportados por los usuarios en las versiones beta. ▪ Con la intención de facilitar el proceso de apropiación del software, se debe incluir en la publicación de las versiones estables no sólo la publicación del código y los paquetes respectivos, sino también el manual de usuarios y el resto de documentación asociada a la versión publicada. ▪ Para facilitar el acceso y uso de las aplicaciones de software se recomienda publicar las versiones estables de éste, no sólo en los sitios web del proyecto sino también en repositorios oficiales, como los utilizados en *Debian*, *Ubuntu*, *Fedora*, entre otros. ▪ Con relación a la publicación del software en repositorios oficiales se sugiere que esta tarea sea ejecutada por personas distintas a los Programadores del software, ya que dicha tarea conlleva tiempo y esfuerzo que puede limitar el avance en el desarrollo del software cuando la misma es llevada a cabo por los programadores. El tiempo y el esfuerzo que involucra la publicación de versiones en los repositorios oficiales se debe a: 1. Dicha publicación tiende a realizarse de manera periódica, al tener nuevas versiones del software (ya sean estas versiones que resulten del agregado o mejora de funcionalidades o por corrección de errores). 2. Por lo general, cada repositorio tiene procedimientos distintos para publicar el software en éstos. En Debian, las personas encargadas de publicar aplicaciones de software en el repositorio de esta distribución se denominan “Mantenedores de paquetes” [4]. ▪ A fin de contar con estadísticas sobre el despliegue y uso de las aplicaciones de software publicadas, se recomienda contar indicadores de descargas de software.
La tabla continúa en la siguiente página	

[4]Información relevante a los procesos que realizan los mantenedores en Debian se encuentra publicada en páginas como:

- https://www.debian.org/doc/manuals/developers-reference/

<table>
<tr><td></td><td>Producto:
Versión estable publicada.</td></tr>
<tr><td></td><td>Responsables:
Líder del proyecto y equipo de desarrollo.</td></tr>
<tr><td colspan="2">Actividad: Aplicación de pruebas de usuarios</td></tr>
<tr><td rowspan="4">Tarea: Probar la versión beta del software correspondiente a la iteración actual.</td><td>Recomendaciones:
Las pruebas que realizan los usuarios son conocidas como pruebas de aceptación, en ellas los usuarios verifican el funcionamiento del software y pueden revisar la interfaz a fin de aprobar o desaprobar elementos de la misma.
Es importante que se le indique a los usuarios la herramienta que pueden utilizar para reportar los errores y/u observaciones respecto a la interfaz, así como indicarles que al reportar un error deben presentar una descripción detallada del mismo. Esta descripción puede incluir una captura de pantalla donde se indica el error en el software o, en el caso de aplicaciones para consola, una captura de la sesión. Para el reporte de errores en aplicaciones de escritorio se pueden utilizar los archivos de bitácoras que se generan en éstas, pues en estos archivos se registran los eventos con niveles configurables de detalle.</td></tr>
<tr><td>Herramienta:
Para reportar errores los usuarios pueden utilizar la herramienta definida en el proyecto para dicha tarea.</td></tr>
<tr><td>Producto:
Reporte de pruebas de usuarios.</td></tr>
<tr><td>Responsables:
Usuarios.</td></tr>
</table>

- https://www.debian.org/doc/manuals/maint-guide/index.en.html

REFERENCIAS

1. J. Alvarez, J. Aguilar, and O. Terán. *Metodología para el Desarrollo Colaborativo de Software Libre (Versión I)*. Fundación Centro Nacional de Desarrollo e Investigación en Tecnologías Libres, 2007. ISBN 978-980-7154-02-4.
2. A. MacIntyre. *After Virtue: A Study in Moral Theory*. University of Notre Dame Press, 1984.
3. J. Alvarez, S. Solé, M. Venegas, and J. Quintero. Aseguramiento de calidad en el desarrollo de software libre, 2013. Documento elaborado en el marco del proyecto Aseguramiento de Calidad en el Desarrollo de Software Libre. Versión disponible en `http://calidad-sl.cenditel.gob.ve/55-2/`.
4. J. Montilva, I. Besembel, and F. Zerpa. Modelado de sistemas usando uml 2.0, 2007.
5. L. Bass, P. Clements, and R. Kazman. *Software Architecture in practice*. Addison-Wesley, 1998.
6. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern – Oriented Software Architecture. A System of Patterns*. John Wiley and Sons, 1996.
7. M. Mcdunnigan. Los factores que determinan la elección del lenguaje de programación usado, s.f. Traducción de Rafael Ernesto Díaz. Versión disponible en `http://www.ehowenespanol.com/factores-determinan-eleccion-del-lenguaje-programacion-usado-info_286132/`.
8. J. Melgoza. 10 criterios para elegir el framework correcto, 2004. Traducción de Rafael Ernesto Díaz. Versión disponible en `http://jonathanmelgoza.com/blog/10-criterios-para-elegir-el-framework-correcto/`.
9. Prueba de concepto, historial de uso, ejemplos, 2012-2014. Versión disponible en `http://centrodeartigos.com/articulos-enciclopedicos/article_93241.html`.
10. G. Rossum. Style guide for python code. python software foundation, 2014. Versión disponible en `http://legacy.python.org/dev/peps/pep-0008/`.
11. D. Lago. Guía de estilo de programación, 2008. Versión disponible en `http://beosman.org/docs/guia-estilo/`.
12. Consejos para escribir buenos casos de uso, 2009. Traducción de Mónica Navarro. Versión disponible en `slideshare.net/kaolong/consejos-para-escribir-buenos-casos-de-uso-10382823`.

13. A. Pérez. Estructuración y especificación de casos de uso, s.f. Documento disponible en https://sites.google.com/site/alfonsoperezr/investigacion/estructuracin-y-especificacin-de-casos-de-uos.

14. S. Cuesta. Patrones de casos de uso, 2007. Documento disponible en http://sg.com.mx/content/view/510.

15. C. Hofmeister, R. Nord, and Soni D. *Applied Software Architecture*. Addison Wesley, 2000.

16. C. Pérez. La importancia del modelo de datos, 2013. Documento disponible en http://liberix.es/blog/la-importancia-del-modelo-de-datos/.

17. V. Madrid and J. De Paz. Servicios web, s.f. Documento disponible en http://www.itescam.edu.mx/principal/sylabus/fpdb/recursos/r100655.PDF.

18. P. Kruchten. *The Rational Unified Process*. Addison Wesley, 1999.

19. G. Booch, J. Rumbaugh, and I. Jacobson. *UML: el lenguaje unificado de modelado*. Addison Wesley, 2000. Traducción, José Sáez Martínez.

20. A. Báez, C. Castañeda, and D. Castañeda. Metodología para el diseño y desarrollo de interfaces de usuario, 2005. Documento disponible en http://pegasus.javeriana.edu.co/~fwj2ee/descargas/metodologia%28v0.1%29.pdf.

21. L. Mateu. Patrones de programación, s.f. Documento disponible en http://users.dcc.uchile.cl/~lmateu/CC10A/Apuntes/patrones/.

22. R. Alvarez. El çódigo espaguetiz los patrones avanzados de programación, s.f. Documento disponible en http://www.tecbolivia.com/index.php/articulos-y-tutoriales-microcontroladores/12-el-qcodigo-espaguetiq-y-los-patrones-avanzados-de-programacion.

23. F. Fernández. phpdocumentor, 2008. Documento disponible en http://www.epsilon-eridani.com/cubic/ap/cubic.php/doc/phpDocumentor---documentacion-para-codigo-PHP-246.html.

24. L. González. Introducción al software de testing, 2012. Documento disponible en https://eseida.wikispaces.com/file/.../Introduccin+al+Software+Testing.p...

25. R. Pressman. *Ingeniería del Software: Un enfoque práctico*. McGraw-Hill, 2002.

26. A. Oré. ¿cómo realizar pruebas funcionales?, 2009. Documento disponible en http://www.calidadysoftware.com/testing/como_realizar_pruebas_funcionales.php.

27. J. Nielsen. *Usability Engineering*. Morgan Kaufmann, 1993.

28. M. Williams, G. Succi, and L. Marchesi. *Traditional and Agile Software Engineering. Ch 8 - Black Box Testing*. Addison Wesley, 2003.