Universidad de las Ciencias Informáticas

Facultad 1

“Herramienta para la administración de repositorios de la Distribución Cubana de *GNU/Linux* Nova”

Trabajo de diploma para optar por el título de Ingeniero en Ciencias Informáticas

Autor: Enmanuel Cristian de la Cruz Mora

Tutores: Ing. Gladys Marsi Peñalver

Ing. Marlon Rodríguez Garcia

La Habana, junio 2017

Frase

"Now, bring me that horizon"

Johnny Deep.

Declaración de autoría.

Declaro por este medio que yo Enmanuel Cristian de la Cruz Mora, con carné de identidad 92011723221 soy el autor principal del trabajo titulado "**Herramienta para la administración de repositorios de la Distribución Cubana de *GNU/Linux* Nova**" y autorizo a la Universidad de las Ciencias Informáticas a hacer uso de la misma en su beneficio, así como los derechos patrimoniales con carácter exclusivo.

Para que así conste, firmo la presente a los ___ días del mes de ___ del año _____.

_____	_____	_____
Firma autor	Firma tutor	Firma tutor

Dedicatoria.

A mi familia, por todo el apoyo que me han brindado a lo largo de mi carrera.

A mi mamá y mi papá, por ser todo lo que más yo quiero en la vida.

A mis hermanos Tata y Ando.

A mi hermanito el Sepi.

Agradecimientos.

A mis padres; por todo el apoyo, la felicidad y el amor que me han dado en todos estos años de mi vida y que gracias a ellos soy quien soy hoy en día.

A mis hermanos Tata y Ando que son los mejores hermanos que uno pueda pedir y han sido mi orgullo en todos estos 25 años. A Yayi y mi sobrina por toda las cosas felices que he pasado junto a ellas y todo el cariño que me han dado hasta ahora.

A tía Berta y Berto por todo el cariño y comprensión que me han dado en estos años.

A mi tía Anita por estar siempre preocupándose y apoyándome a lo largo de estos años .

A mi otra mamá la Flaki y a mis hermanitos Leo y Félix por toda la comprensión y el cariño que he tenido de ellos desde el principio.

A mis tutores; Gladys y Marlon por hacer que este sueño se hiciera realidad y no darse nunca por vencidos conmigo.

A todas las amistades de CESOL que estuvieron conmigo durante toda la tesis: Michel, Yileni, Neyvis, Yisel, Manuel Alejandro, Manuel Peiso, Yanet, Yaiselis, Ruben, Jorge, Luis, Juan y a todos los profesores que hicieron posible que lograra llegar a este día.

A todas las amistades que he hecho en estos 6 años de carrera por todos los buenos momentos que he pasado junto a ellos, Adrián, Oberman, Manano, Raúl, Laura, Bia, Ada, Stephany, Betsy, Camps y Edel.

A todas las hermanitas que quiero: Wilbia, Patricia y Lisbeth.

A Rubén y Asney que son los mejores amigos que uno pueda tener y con los que he compartido gratos momento de Universidad y siempre han estado ahí para mi en las buenas y en las malas y a Ale que me enseño que el que no coge un buen doble no sabe lo que es la vida.

Al piquete del 110 103, buenos amigos que aunque estuvimos solo un año compartiendo conmigo para mi fue como si los conociera desde siempre: Marbier, Yobal, el Dani y Adrián.

A mis hermanitos Amaury y el Yoja, que aunque uno se encuentra lejos estuvieron conmigo todos estos seis años y a lo largo de la tesis apoyándome y ayudando en todo lo que pudieron.

A mi hermanaso el Sepi que siempre pensé que aun me faltaba un hermanito más y lo encontré aquí en la universidad, le agradezco todo el apoyo y todo el aguante que tuvo que tener conmigo durante todos estos años de la carrera y que se que aún me ha tenido que soportar, gracias Cuco.

En fin a todas las personas que hicieron posible que se cumpliera mi sueño de convertirme en Ingeniero de Ciencias Informáticas en la Universidad que quise desde pequeño, gracias por todo.

Resumen.

La gestión de repositorios cumple un factor importante dentro de la comunidad de usuarios que utilizan los sistemas *GNU/Linux*. En la Distribución Cubana de *GNU/Linux* Nova se gestionan los repositorios utilizando la herramienta *Reprepro*, esta herramienta aunque realiza la gestión de los repositorios no cumple con todos las necesidades del usuario. Reprepro solamente permite una versión de un paquete dentro del repositorio y no existe la posibilidad de realizar r*ollbacks y* no utiliza *snapshots*. La presente investigación propone la realización de una herramienta web, con el objetivo de realizar la gestión de los repositorios de forma centralizada de la Distribución Cubana de *GNU/Linux* Nova. Para ello se realiza un estudio de las herramientas más utilizadas para la gestión de repositorios, se define como metodología AUP-UCI, la cual guiará el proceso de desarrollo de *software.* Para la implementación de la solución propuesta se hará uso del *Framework* de desarrollo *Node.js* con el servidor web *Express,* como herramienta de modelado Visual Paradigm y como lenguajes de programación se utilizó Javascript, HTML5 y CSS3*.* Con todo lo antes planteado *s*e obtuvo una herramienta que permite crear, modificar, listar y eliminar los repositorios además de añadir, listar y eliminar paquetes de los mismos.

Palabras claves: *framework,* repositorio, rollbacks, paquetes, snapshots.

Índice de contenido

Índice de tablas

Índice de ilustraciones

Introducción.

Las Tecnologías de la Información y las Comunicaciones (TIC) tienen un factor importante dentro del campo económico y sociocultural de los países convirtiéndose en una de las principales vías de desarrollo de los mismos. La calidad del uso de las tecnologías informáticas y la información obtenida a partir de estas, varía según el sistema operativo utilizado. Los sistemas operativos *GNU/Linux*[1] surgen como una alternativa a otros sistemas que restringían funcionalidades, ya sea por la imposibilidad de realizar la tarea solicitada o por los aletargados ciclos de respuesta del proveedor. Las distribuciones de *GNU/Linux*, han alcanzado un alto grado de madurez y muchos seguidores al código abierto se han empeñado en lograr sistemas operativos que cumplan con las necesidades y demandas de los usuarios. Con estos sistemas se puede trabajar en diferentes tipos de proyectos, ya que están desarrollado como una robusta y eficaz herramienta de trabajo. Estas características han dado como resultado que en los últimos años *GNU/Linux* haya sido el sistema operativo elegido que desplazó a *Microsoft Windows*[2] de muchas computadoras. Debido a su facilidad de instalación y uso es compatible con casi cualquier tipo de dispositivo que requiera de un sistema operativo. Este gran cambio producido en la última década, ha logrado posicionar a *GNU/Linux* como uno de los sistemas operativos más elegidos a nivel profesional, por ser realmente estable, confiable y seguro, además de ser libre (1).

En Cuba el proceso de utilización ordenado y masivo de las TIC para satisfacer las necesidades de información y conocimiento de todas las personas y esferas de la sociedad ha tenido un gran auge. Este proceso busca lograr eficiencia y eficacia, que permitan una mayor generación de riquezas y hagan

---

1

Es uno de los términos empleados para referirse a la combinación del núcleo o kernel libre similar a Unix denominado Linux con el sistema operativo GNU.

2 Conocido generalmente como Windows o MS Windows, es el nombre de una familia de distribuciones de software para PC, smartphone, servidores y sistemas empotrados, desarrollados y vendidos por Microsoft y disponibles para múltiples arquitecturas, tales como x86 y ARM.

sustentable el aumento sistemático de la calidad de vida de los ciudadanos en su desempeño familiar, laboral, educacional, cultural y social, sobre una política preferentemente orientada al uso social e intensivo de los recursos TIC, para extender sus beneficios a la mayor parte posible de la población y las instituciones. Cuba en lo últimos años ha venido realizando una gran labor en el campo de la informatización de la sociedad. En la Universidad de las Ciencias Informáticas (UCI), en el Centro de Soluciones Libres (CESOL) se desarrolló la Distribución Cubana de *GNU/Linux* Nova, que tiene como objetivo la migración al código abierto y el *software*[3] libre en Cuba para garantizar la independencia tecnológica.

La Distribución Cubana de *GNU/Linux* Nova al igual que todos los sistemas basados en *GNU/Linux* usan repositorios para instalar los programas necesarios para el usuario. Los repositorios son grandes bancos de datos o servidores que alojan las aplicaciones que el sistema necesita, entre ellos, paquetes nuevos y actualizaciones que se instalan mediante un manejador de paquetes. Actualmente en Nova se gestionan los repositorios utilizando *Reprepro*. Esta herramienta se utiliza para gestionar un repositorio de paquetes. Genera un fichero de tipo *index*[4] (*Release* y opcionalmente *Release.gpg*), por lo que las herramientas como *apt*[5] saben lo que está disponible y de dónde obtenerlo. También puede hacer espejos parciales de repositorios remotos, incluyendo la fusión de múltiples fuentes y automáticamente (si se solicita explícitamente) la eliminación de paquetes que ya no están disponibles en el origen. Las principales deficiencias de utilizar *Reprepro* son: solamente permite una versión de un paquete dentro del repositorio. Si se tienen diferentes aplicaciones que necesiten un mismo paquete pero en diferentes versiones para su uso, la herramienta reprepro no puede realizar esta acción. No existe la posibilidad de realizar *rollbacks*[6] *y* no utiliza *snapshots*[7], o sea, no tiene copia de seguridad de los ficheros almacenados tal y como fueron capturados en el pasado. Si existe un problema en el sistema y deja de funcionar una aplicación debido a

3 Conjunto de programas y rutinas que permiten a la computadora realizar determinadas tareas.

4 Es un archivo de computadora con un índice que permite un fácil acceso aleatorio a cualquier registro dada su clave de archivo.

5 Advanced Packaging Tool (Herramienta de empaquetado avanzada), es una herramienta informática de gestión de paquetes desarrollada por Debian.

6 En tecnologías de base de datos, un rollback o reversión es una operación que devuelve a la base de datos a algún estado previo.

que un paquete se encuentra roto o no existe el paquete dentro del repositorio no hay copia de seguridad de ficheros anteriores a este.

Partiendo de esta situación problemática surge el siguiente **problema de investigación**: ¿Cómo mejorar la seguridad en la administración de los repositorios de software de la Distribución Cubana de *GNU/Linux* Nova?. Se define como **objeto de estudio** el proceso de administración de los repositorios en los sistemas *GNU/Linux* enmarcado en el **campo de acción** los repositorios de la Distribución Cubana de *GNU/Linux* Nova. Por lo que se formula como **objetivo general** desarrollar una herramienta que permita la seguridad de la administración centralizada de los repositorios de la Distribución Cubana de *GNU/Linux* Nova.

Para darle cumplimiento al objetivo general planteado, el mismo se dividió en los siguientes **objetivos específicos.**

- Analizar las herramientas de administración de repositorios enfocándose en las más utilizadas actualmente.
- Diseñar una herramienta que permita la administración de los repositorios de la Distribución Cubana *GNU/Linux* Nova.
- Implementar la herramienta para la administración de los repositorios de la Distribución Cubana *GNU/Linux* Nova.
- Realizar pruebas a la herramienta para la administración de los repositorios de la Distribución Cubana *GNU/Linux* Nova.

Como **preguntas científicas** se plantean:

¿Cuáles son las tendencias actuales relacionadas con las herramientas para la administración de repositorios?

---

7 Snapshot o copia instantánea de volumen. Es una función de algunos sistemas que realizan copias de seguridad de ficheros tal y como fueron almacenados en el pasado.

¿Qué tecnologías, herramientas y metodología se requieren utilizar para el desarrollo de la herramienta para la administración de repositorios de Nova?

¿Cuáles son los elementos a tener en cuenta para el análisis y diseño de la herramienta para la administración de los repositorios de Nova?

¿Qué resultado se obtendrá al validar la herramienta a desarrollar?

Para cumplir con los objetivos específicos se proponen las siguientes **tareas de investigación**:

- Análisis de los principales conceptos relacionados con la investigación.
- Investigación sobre los sistemas homólogos y las herramientas a utilizar para el desarrollo de la solución.
- Definición de los requisitos funcionales y no funcionales.
- Implementación de las funcionalidades de la herramienta.
- Aplicación de las pruebas a la herramienta desarrollada.

Se espera como resultado una herramienta para la administración de los repositorios de los sistemas *GNU/Linux*.

Para el proceso de desarrollo de la investigación se emplearon los siguientes métodos:

Técnica:

La tormenta de idea fue el método empírico utilizado en el proceso de investigación para la determinación de las restricciones de *software* e implementación que se debe tener en cuenta en el proceso de desarrollo de la herramienta.

Métodos científicos:

El método **inductivo-deductivo** permite llegar a proposiciones generales a partir de hechos aislados que confirman la teoría o a partir de estas teorías arribar a conclusiones sobre casos particulares que se verifican en la práctica (2). Este método fue usado para, a partir del análisis de las herramientas existentes que realizan la gestión de repositorios, seleccionar las características que va a poseer la herramienta a desarrollar de forma tal que realice correctamente la gestión de los repositorios.

El método **histórico-lógico** plantea que se debe estudiar la trayectoria real de los fenómenos y acontecimientos en el transcurso de una etapa o período analizando las leyes generales del funcionamiento y desarrollo del fenómeno, estudiando su esencia (2). Fue empleado con el objetivo de verificar cómo evolucionan teóricamente las herramientas que realizan la gestión de los repositorios y de este modo poder realizar una selección de las técnicas, herramientas y los algoritmos que se van a utilizar para desarrollar .

El método **analítico-sintético** consiste en la extracción de las partes de un todo, con el objetivo de estudiarlas y examinarlas por separado para luego sintetizarla y tomar los elementos más importantes (2). Este método permitió el estudio de diferentes fuentes bibliográficas para extraer los elementos más importantes que se relacionan con las herramientas para la administración de los repositorios de distribuciones *GNU/Linux*.

El método **revisión bibliográfica** permite identificar las fuentes arbitradas[8], identificar y comprender las ideas principales y sintetizar el argumento central de cada sección a través de la supresión y generalización de ideas principales (3). Dicho método fue utilizado para determinar las fuentes y referencias bibliografías óptimas y actualizadas para la elaboración de la investigación.

El presente documento se encuentra dividido en tres capítulos.

El capítulo 1, **Fundamentación Teórica:** describe los principales conceptos asociados al problema de investigación planteado. Se realiza un análisis de las principales características y deficiencias de cada herramienta encontrada y finalmente, se seleccionan las herramientas que van a ser utilizadas para el desarrollo de la solución y se selecciona la metodología de desarrollo que va a guiar todo el proceso de

8 Aquella que cuenta con un comité editorial que revisa con un procedimiento de jueces la pertinencia, relevancia, originalidad y rigor metodológico de una publicación.

construcción de la herramienta para la administración de los repositorios en la Distribución Cubana *GNU/Linux* Nova.

El capítulo 2, **Análisis y Diseño de la Solución:** se realizan las fases de **inicio y ejecución** de la metodología descrita en el capítulo 1, esta última fase continúa en el capítulo 3. Se realiza la extracción de requisitos funcionales y no funcionales de la herramienta a desarrollar, se modela la estructura y funcionamiento de la solución a través de los diagramas de componentes y paquetes. Además, se seleccionan los patrones de diseño que serán usados para resolver la problemática.

En el capítulo 3, **Implementación y Pruebas de la Solución:** se continúa con la fase de **ejecución** de la metodología escogida en el capítulo 1 y se realiza la tercera fase de esta metodología para el desarrollo de la herramienta, se hace el análisis de las funciones identificadas como vulnerables. Se explican los distintos tipos de pruebas de *software* que se van a utilizar para comprobar el correcto funcionamiento de la solución, así como los resultados que arrojaron la utilización de las mismas y se muestra el impacto social que va a tener la investigación.

Capítulo 1. Fundamentación teórica.

Los repositorios cumplen con un papel muy importante dentro de los sistemas *GNU/Linux* ya que son los encargados de contener todos los paquetes que estos sistemas necesitan para su correcto funcionamiento, además tienen actualizaciones de dichos paquetes para que el usuario si lo desea siempre tenga actualizado todos los programas instalados en su ordenador y a su vez tener versiones nuevas de estos paquetes ya que muchos programas dependen de estas para su correcto funcionamiento. En este capítulo se definen los principales conceptos relacionados con la gestión de repositorios así como las herramientas a utilizar para la implementación de la solución.

1.1. Conceptos y definiciones asociados al dominio del problema.

- **Repositorios institucionales**: son los creados por las propias organizaciones para depositar, usar y preservar la producción científica y académica que generan. Supone un compromiso de la institución con el acceso abierto al considerar el conocimiento generado por la institución como un bien que debe estar disponible para toda la sociedad (4).
- **Repositorios temáticos**: son los creados por un grupo de investigadores, una institución, etc. que reúnen documentos relacionados con un área temática específica (4).
- **Repositorios de datos**: repositorios que almacenan, conservan y comparten los datos de las investigaciones (4).

Paquetes: es un grupo de uno o más archivos que son necesarios tanto para la ejecución de un programa de computadora o como para agregar características a un programa ya instalado en la computadora o en una red de computadoras (5).

Gestor de paquetes: es un sistema que mantiene un registro del *software* que está instalado en el ordenador y permite instalar *software* nuevos, actualizarlos a versiones más recientes o eliminar *software*

de manera sencilla. La labor de un gestor de paquetes es la de presentar una interfaz que asista al usuario en la tarea de administrar el conjunto de paquetes que están instalados en su sistema (6).

***API*:** es una sigla que procede de la lengua inglesa y que alude a la expresión *Application Programming Interface* (cuya traducción es Interfaz de Programación de Aplicaciones). El concepto hace referencia a los procesos, las funciones y los métodos que brinda una determinada biblioteca de programación a modo de capa de abstracción para que sea empleada por otro programa informático (7).

1.2. Estudio de sistemas homólogos.

El análisis de las particularidades de las herramientas similares para administrar los repositorios de los sistemas *GNU/Linux*, permite conocer características que, por su relevancia, pueden ser fundamentales para la propuesta a desarrollar o servir como objeto de estudio para realizar una selección acertada de estándares, herramientas y tecnologías a utilizar para su construcción. Para definir las herramientas de gestión de repositorios que se tendrán en cuenta para el estudio de homólogos se deben tener en cuenta las características fundamentales que presentan los repositorios de las distribuciones de Nova. Dichas herramientas deberán ser capaces de gestionar repositorios de código binario (.deb y .udeb), gestionar los paquetes (adicionar y eliminar) y modificar la estructura del repositorio.

Algunas de las principales herramientas que realizan la administración de repositorios son, *Dak, Mini-dak, PPA, Reprepro* y *Aptly,* aunque *Dak y Mini-dak* presentan limitaciones que hacen que estas herramientas no cumplan los requerimientos necesarios para darle solución al problema de la investigación. Estas limitantes serían(6):

Limitantes de utilizar *Dak*.

- Depende de *Python*[9] y *PostgreSQL*[10].

9 Es un lenguaje de programación multiparadigma ya que soporta orientación a objetos, programación imperativa y, en menor medida, programación funcional.

10 Es un Sistema de gestión de bases de datos relacional orientado a objetos y libre.

- Falta de documentación.
- Diseñado para grandes repositorios.

Limitantes de utilizar *Mini-dak*.

- Lento en grandes repositorios debido a que no usa una base de datos real.
- Todavía tiene algunas peculiaridades que deben arreglarse.

Por tal motivo se escogen y se analizan como herramientas principales para la administración de repositorios de acuerdo a los criterios antes planteados para la presente investigación los Archivos de Paquetes Personales (*PPA*), *Reprepro* y la herramienta *Aptly*. A continuación se describen cada una de estas herramientas así como su funcionamiento y características principales.

1.2.1. Archivo de Paquetes Personales.

Un Archivo de Paquetes Personales o como se conoce por sus siglas en ingles *Personal Package Archive* (PPA) es un repositorio de *software* para cargar paquetes de código fuente que se creará y publicará como un repositorio de herramientas de empaquetado avanzado en *Launchpad*[11]. Mientras que el término se utiliza exclusivamente en *Ubuntu*[12], el anfitrión de *Launchpad Canonical* prevé la adopción más allá de la comunidad de *Ubuntu*.

Utilizando un archivo de paquetes personales (PPA), se puede distribuir *software* y actualizaciones directamente a los usuarios de *Ubuntu*. Creando un paquete fuente y cargándolo en *Launchpad*, este construirá binarios y luego los alojará en su propio repositorio apt. Esto significa que los usuarios pueden instalar sus paquetes de la misma manera que instalan paquetes estándar de *Ubuntu* y automáticamente

11 Es una plataforma de colaboración de software que proporciona, seguimiento de errores, revisiones de código, construcción y alojamiento de paquetes en Ubuntu.

12 Filosofía africana cuyo significado refleja **“Humanidad hacia otros”**. En términos informáticos, es un sistema operativo enfocado en dispositivos electrónicos, construido a partir del núcleo Linux (Linux Kernel). Ubuntu es una distribución de GNU/Linux.

recibirán actualizaciones a medida que las hagan. Cada usuario y equipo en *Launchpad* puede tener uno o más PPAs, cada uno con su propia *URL*[13]. Los paquetes que se publiquen en el PPA permanecerán allí hasta que se eliminen, sean reemplazados por otros paquetes o la versión de Ubuntu contra la que están construidos se vuelva obsoleta.

El tamaño y límite de transferencia de cada PPA es de 2GB de espacio en el disco duro. Antes de poder empezar a usarla, ya sea propio del usuario o pertenezca a un equipo, se debe activar en la página de perfil del usuario o en la página de resumen del equipo. Si ya se tiene uno o más PPA, también es donde se podrá crear archivos adicionales. *Launchpad* genera una clave única para cada PPA y la usa para firmar cualquier paquete incorporado en ese repositorio. Esto significa que los usuarios que descargan o instalan paquetes desde sus repositorios personales pueden verificar su fuente. Cuando ya no se necesite un paquete, se puede eliminar. Esto elimina todos los paquetes de la PPA y elimina el repositorio de *ppa.launchpad.net.* Se tendrá que esperar hasta una hora para poder recrear un paquete con el mismo nombre (8).

Añadir repositorios extra en la distribución *Linux* que el usuario utilice es muy útil, ya sea para actualizar algún *software* o paquete que la distribución no actualiza o mantiene o bien porque ese paquete no está disponible de serie en los repositorios del usuario y se necesita añadirlo para poder instalarlo. Una de las ventajas más importante de utilizar un PPA es estar siempre actualizado en un paquete/*software* concreto, automatizar el proceso de instalación sin tener que recurrir a la instalación manual mediante código fuente o tener la capacidad de instalar los paquetes necesarios ajenos al *software* en cuestión de modo automático. Aunque utilizar PPA para administrar los repositorios también tiene sus desventajas y una de las más comunes es que, al añadir una PPA para mantener actualizado la última versión de un paquete puede implicar que aparezcan algunas incompatibilidades entre distribución o paquete y en algunas ocasiones la última versión del paquete en cuestión presenta algunos *bugs*[14] o inconsistencias que le hacen ser inestable (9).

13 Uniform Resource Locator (Localizador Uniforme de Recursos). Se trata de la secuencia de caracteres que sigue un estándar y que permite denominar recursos dentro del entorno de Internet para que puedan ser localizados.

14 Problema en un programa de computador o sistema de *software* que desencadena un resultado indeseado.

1.3.2. Reprepro.

Reprepro es una herramienta para gestionar un repositorio de paquetes *Debian* (.deb, .udeb, .dsc, entre otras). Almacena archivos que se inyectan manualmente o se descargan de algún otro repositorio (parcialmente) reflejado en una agrupación / jerarquía. Los paquetes administrados y las sumas de comprobación de los archivos se almacenan en una base de datos libdb4.3 (o libdb4.4 o libdb3, dependiendo de qué *Reprepro* haya sido compilado), por lo que no se necesita ningún servidor de base de datos. Se admite la comprobación de firmas de repositorios duplicados y la creación de firmas de los índices de paquetes generados (10).

Funcionalidades.

La configuración de un repositorio utilizando *Reprepro* se realiza escribiendo algunos archivos de configuración en un directorio, este archivo es el subdirectorio ***conf*** del directorio base del repositorio a menos que especifique --***confdir***. Si existe este archivo, *Reprepro* considerará cada línea una opción de línea de comandos adicional. Los argumentos deben estar en la misma línea después de un signo igual. Las opciones especificadas en la línea de comandos tienen prioridad. Este archivo es el de la configuración principal y el único que se necesita en todos los casos. Enumera las distribuciones que contienen este repositorio y sus propiedades. *Reprepro* puede generar archivos de contenidos llamados ***dists / CODENAME / Contents-ARCHITECTURE.gz*** listando todos los archivos de todos los paquetes binarios disponibles para la arquitectura seleccionada en esa distribución y a qué paquete pertenecen. *Reprepro* puede obtener paquetes de otros repositorios utilizando los métodos de apt. También puede incluir paquetes binarios de *Debian* (.deb y .udeb) e incluir el paquete fuente de *Debian* (.dsc, incluyendo otros archivos como .orig.tar.gz, .tar.gz y .diff.gz) en la distribución especificada, aplicando la información de reemplazo y con la configuración por defecto tomando todos los valores no dados y adivinables. Además, tiene dentro del archivo de configuración un campo nombrado **update** (actualizar) que describen que reglas de actualización se utilizan para esta distribución, además de un campo **pull** (halar) que describe las reglas de extracción que se utilizarán para la distribución. Las reglas de extracción son como las reglas de actualización, pero obtienen su información de otras distribuciones y no de fuentes externas.

Las variables de entorno en siempre se sobrescriben mediante opciones de línea de comandos, pero las opciones de sobrescritura se establecen en el archivo de opciones, Incluso cuando el archivo de opciones es obviamente analizado después de las variables de entorno, ya que el entorno puede determinar el lugar del archivo de opciones (11).

Ejemplo de alguna variables de entorno son las siguientes:
REPREPRO_BASE_DIR : el directorio de esta variable se utiliza en lugar del directorio actual, si no se proporcionan opciones -b o --basedir.
REPREPRO_CONFIG_DIR : el directorio de esta variable se utiliza cuando no se proporciona --confdir.

Reprepro presenta como objetivo poner todos los archivos en una agrupación jerarquía coherente. Por lo tanto, no puede garantizar que los archivos tendrán la misma ruta de parentesco que en el repositorio original (especialmente si no tienen ningún grupo). También crea los archivos de índice de sus propios índices. Si bien esto lleva a un repositorio ordenado y posibles ahorros de espacio en disco, las firmas de los repositorios que refleja no se pueden usar para autenticar el espejo, pero tendrá que firmar (o decirle a *Reprepro* para firmar por el usuario) el resultado. Si bien esto es perfecto cuando sólo refleja algunas partes o paquetes específicos o también tiene paquetes locales que necesitan firma local de todos modos, *Reprepro* no es una herramienta adecuada para crear un espejo completo que se puede autenticar sin agregar la clave de este repositorio. Además, hace todo el cálculo de nombres de archivos para guardarlos como, la contabilidad de los que hay y los que se necesitan y así sucesivamente. No puede ser cambiado o desactivado. El usuario puede colocar archivos donde *Reprepro* los esperará y los utilizará si su md5sum coincide. Pero *Reprepro* no es adecuado si se desea que los archivos estén fuera del ***pool*** (piscina) o en otros lugares.

1.3.3. Aptly.

Aptly es una herramienta para la gestión de repositorios de *Debian*. Le permite crear espejos de repositorios remotos, administrar repositorios de paquetes locales, tomar *snapshot* y extraer nuevas versiones de paquetes junto con dependencias. *Aptly* realiza espejos con *snapshot* de los repositorios con actualizaciones de seguridad incorporadas para hacer que la instalación del paquete sea consistente en

todos los hosts[15]. Además, permite al usuario crear su propio repositorio remoto lo cual permite ahorrar el ancho de banda si se tiene un gran número de servidores, así como que no dependen de los servicios remotos para funcionar correctamente con el fin de gestionar las piezas críticas de la infraestructura. El objetivo de *Aptly* es establecer repetibilidad y cambios controlados en entornos de paquetes. Produce adecuadamente un conjunto fijo de paquetes en el repositorio, de modo que la instalación y actualización del paquete se convierte en determinista. Al mismo tiempo, es capaz de realizar cambios controlados y finos en el contenido del repositorio. Convenientemente le permite la transición de su entorno de paquete a una nueva versión, o regresar (*rollback*) a una versión anterior. *Aptly* tiene varias entidades centrales: Espejo (*mirror*) del repositorio remoto, consiste en metadatos, lista de paquetes y archivos de paquetes de repositorios locales, se compone de metadatos, paquetes y archivos, los paquetes se pueden agregar y quitar fácilmente. Los *snapshot* son listas inmutables de paquetes, bloque básico para implementar repetibilidad y cambios controlados publicados y los repositorios son representación publicadas de *snapshot* generados por *Aptly* o repositorios locales, listos para ser consumidos por las herramientas apt. Lo antes planteado se puede ver en la Ilustración 1.

15 Es un equipo u otro dispositivo que se comunica con otros hosts en una red. Los hosts de una red incluyen clientes y servidores que envían o reciben datos, servicios o aplicaciones.

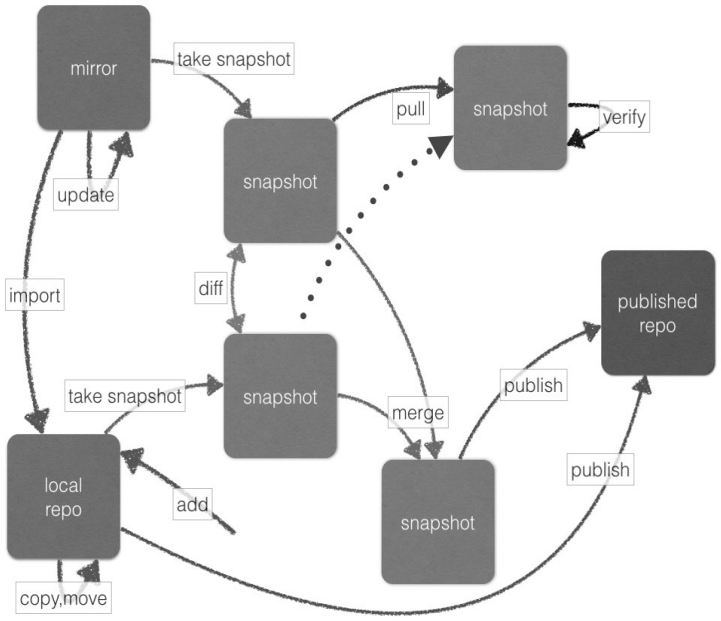

Ilustración 1: Esquema de los comandos y transiciones entre las entidades.

En ocasiones los usuarios desean crear sus propios paquetes, ya sea, un software propio o una versión remendada de paquetes oficiales. Históricamente, las herramientas para crear repositorios de paquetes eran complejas y requerían una configuración compleja. *Aptly* realiza las cosas de manera diferente: la gestión de los repositorios locales es fácil, se le pueden hacer snapshot como repositorios remotos y los paquetes de los repositorios locales pueden mezclarse con los paquetes de los espejos (en los snapshot) (12).

Características de *Aptly*.

Reflejar repositorios remotos:

- Repositorios regulares y planos.

- Se soportan servidores *HTTP*[16] o *HTTPS*[17] y *FTP*[18].
- Espejo sólo especifica las arquitecturas / componentes.
- Espejos parciales (con filtros en los paquetes).
- Espejo de búsqueda de paquetes que coinciden con la consulta.
- Soporte de paquetes .udeb (instalador de *Debian*).

Manejo de repositorios locales:

- Cualquier número de repositorios locales.
- Los paquetes se podrían agregar de archivos o por exploración del directorio.
- Los paquetes fuente extraen automáticamente todos los archivos relacionados.
- Mover y copiar paquetes entre repositorios.
- Importar paquetes de espejos.
- Eliminación de paquetes que coinciden con la condición.

Conjunto de paquetes (almacenamiento de archivos de paquetes internos):

- Paquetes de espejos y repositorios locales que se almacenan de forma duplicada.
- El archivo de paquete se mantiene en el conjunto de paquetes hasta que haya al menos una sola referencia.

16 Abreviatura de la forma inglesa *Hypertext Transfer Protocol (*protocolo de transferencia de hipertextos), que se utiliza en algunas direcciones de internet.

17 Hypertext Transfer Protocol Secure (protocolo de transferencia segura de hipertextos) es una combinación del protocolo HTTP y protocolos criptográficos. O sea es la versión segura de HTTP.

18 Es uno de los diversos protocolos de la red Internet, concretamente significa File Transfer Protocol (Protocolo de Transferencia de Ficheros) y es el ideal para transferir grandes bloques de datos por la red.

- Buscar y mostrar detalles de paquetes en la piscina.
- Manejo de paquetes duplicados.

***Snapshot* para espejos y repositorios locales:**

- Crear *snapshot* de espejos y repositorios locales.
- Los *snapshot* son inmutable.
- Combinando varios *snapshot* en uno.
- Filtrar *snapshots* que producen un nuevo *snapshot.*
- Tirando paquetes que emparejan la consulta de un *snapshot* a otro, produciendo un nuevo *snapshot.*
- Comprobación de *snapshot* de dependencias insatisfechas.
- Calcular la diferencia entre *snapshot.*
- Buscar *snapshot* de paquetes que coincidan con la consulta.

Publicación de *snapshot* y repositorios locales:

- Publicar *snapshot* creados a partir de espejos o repositorios locales.
- Publicar repositorios locales directamente.
- Publicación de componentes múltiples.
- Conmutación atómica de instantáneas publicadas.
- Actualización atómica de repositorios locales publicados.

- Visualización de gráficas de dependencias entre espejos, repositorios locales, *snapshot* y repositorios publicados.
- Servir rápidamente repositorios publicados a través de *HTTP*.

1.3.4. Análisis de las principales herramientas para la gestión de repositorios.

A continuación se muestra una tabla donde se realiza una comparación entre las herramientas que realizan la gestión de repositorios y se tiene en cuenta como criterio de comparación sus características más fundamentales.

Características	*PPA*	*Reprepro*	*Aptly*
Versiones dentro del repositorio.	No permite	No permite	Permite
Rollbacks.	No permite	No permite	Permite
Snapshots.	No utiliza	No utiliza	Utiliza
Gestión de repositorios.	No seguro	Seguro	Seguro
Interfaz gráfica.	No presenta	No presenta	No presenta

Tabla 1: Comparación de los sistemas homólogos.

Las herramientas analizadas en la tabla anterior corresponden con las alternativas más utilizadas para la gestión de repositorios. Dicha tabla arrojó que la herramienta *PPA* y la herramienta *Reprepro* no permiten versiones dentro del repositorio, no utilizan ni permiten *rollbacks* ni *snapshots* y no presentan una interfaz gráfica para la interacción con el usuario. Por otro lado la herramienta *Aptly* si permite las características anteriormente expuestas aunque tampoco presenta una interfaz gráfica con la que el usuario pueda interactuar, a pesar de esto se escoge dicha herramienta para la creación de la propuesta de solución. El estudio de estas herramientas permitió un mayor conocimiento de las prácticas utilizadas para la gestión de los repositorios.

1.4. Metodología de desarrollo de software.

Según la Real Academia de la Lengua Española una metodología es un conjunto de métodos que se siguen en una investigación científica o en una exposición doctrinal. También una metodología de desarrollo de *software* es un conjunto de pasos y procedimientos que debe seguirse para desarrollar un *software*. Una metodología está compuesta por varias etapas, las tareas que se realizan en las etapas, las restricciones que deben aplicarse, las técnicas y herramientas que se deben emplear y la forma de controlar y gestionar el proyecto (13).

Existen dos tipos de metodologías, **tradicionales o robustas** y **ágiles**. Las metodologías **tradicionales o robustas** están pensadas para el uso exhaustivo de documentación durante todo el ciclo de vida del proyecto y las metodologías **ágiles** se basa en dos aspectos puntuales, el retrasar las decisiones y la planificación adaptativa; permitiendo potenciar aún más el desarrollo de software a gran escala.

Retrasar las decisiones y Planificación Adaptativa es el eje en cual gira la metodología ágil, el retrasar las decisiones tan como sea posible de manera responsable será ventajoso tanto para el cliente como para la empresa, lo cual permite siempre mantener una satisfacción en el cliente y por ende el éxito del producto, las principales ventajas de retrasar las decisiones son:

- Reduce el número de decisiones de alta inversión que se toman.
- Reduce el número de cambios necesario en el proyecto.
- Reduce el coste del cambio.

La planificación adaptativa permite estar preparados para el cambio ya que se ha introducido en el proceso de desarrollo del *software*, además hacer una planificación adaptativa consiste en tomar decisiones a lo largo del proyecto, se estará transformando el proyecto en un conjunto de proyectos pequeños. Esta planificación a corto plazo permitirá tener *software* disponible para nuestros clientes y además ir aprendiendo de la retroalimentación para hacer la planificación más sensible, ya sea ante inconvenientes que aceleren o retrasen el producto (14).

Las ideas fundamentales que presentan las metodologías ágiles son:

- Los individuos y las interacciones entre ellos son más importantes que las herramientas y los procesos empleados.
- Es más importante crear un producto *software* que funcione que escribir documentación exhaustiva.
- La colaboración con el cliente debe prevalecer sobre la negociación de contratos.
- La capacidad de respuesta ante un cambio es más importante que el seguimiento estricto de un plan.

Dentro de las principales metodologías ágiles se encuentran XP (eXtreme Programming), Scrum, AUP y actualmente fue desarrollada en la Universidad de las Ciencias Informáticas una metodología basada en AUP nombrada AUP-UCI. Estas metodologías ponen de relevancia que la capacidad de respuesta a un cambio es más importante que el seguimiento estricto de un plan. Para muchos clientes esta flexibilidad será una ventaja competitiva y porque estar preparados para el cambio significa reducir su coste.

Para el desarrollo de la solución propuesta se seleccionó *AUP-UCI* debido a que es una metodología ágil y está basada en *AUP*, de forma tal que se adapta al ciclo de vida definido para la actividad productiva de la UCI. Además de que está centrada en potenciar las relaciones interpersonales como clave del éxito. A través de su utilización se promueve el trabajo en equipo, predominando el aprendizaje de los desarrolladores y un buen clima de trabajo. Se basa en la realimentación continua entre el cliente y el equipo de desarrollo, la comunicación fluida entre todos los participantes, la simplicidad en las soluciones implementadas y el coraje para enfrentar los cambios(15).

Descripción de las fases.

AUP-UCI presenta tres fases las cuales son:

Inicio: Durante el inicio del proyecto se llevan a cabo las actividades relacionadas con la planeación del proyecto. En esta fase se realiza un estudio inicial de la organización cliente que permite obtener

información fundamental acerca del alcance del proyecto, realizar estimaciones de tiempo, esfuerzo y costo y decidir si se ejecuta o no el proyecto.

Ejecución: En esta fase se ejecutan las actividades requeridas para desarrollar el *software*, incluyendo el ajuste de los planes del proyecto considerando los requisitos y la arquitectura. Durante el desarrollo se modela el negocio, obtienen los requisitos, se elaboran la arquitectura y el diseño, se implementa y se libera el producto.

Cierre: En esta fase se analizan tanto los resultados del proyecto como su ejecución y se realizan las actividades formales de cierre del proyecto.

Descripción de las disciplinas.

Para el ciclo de vida de los proyectos de la UCI se decide tener siete disciplinas. Se consideran los flujos de trabajos: **Modelado de negocio**, **Requisitos y Análisis** y **Diseño** disciplinas de la metodología. Se mantiene la disciplina **Implementación** al igual que en AUP, en el caso de la disciplina Prueba se divide en tres disciplinas: **Pruebas Internas**, de **Liberación** y **Aceptación**. Las restantes tres disciplinas que presenta *AUP* asociadas a la parte de gestión se cubren con las las áreas de procesos que define *CMMI-DEV* v1.3[19] para el nivel 2, serían CM (Gestión de la configuración), PP (Planeación de proyecto) y PMC (Monitoreo y control de proyecto).

Escenarios para la disciplina Requisitos.

A partir de que el Modelado de negocio propone tres variantes a utilizar en los proyectos (Casos de Uso del Negocio, Descripción de Proceso de Negocio o Modelo Conceptual) y existen tres formas de encapsular los requisitos (Casos de Uso del Sistema, Historias de usuario, Descripción de requisitos por proceso), surgen cuatro escenarios para modelar el sistema en los proyectos. Para modelar la propuesta de solución se tomo el escenario 4 dadas sus características. Es aplicable a proyectos que hayan evaluado el negocio a informatizar y como resultado obtengan un negocio muy bien definido. El cliente estará siempre acompañando al equipo de desarrollo para convenir los detalles de los requisitos y así

19 Una guía para aplicar las mejores prácticas en una entidad desarrolladora. Estas prácticas se centran en el desarrollo de productos y servicios de calidad.

poder implementarlos, probarlos y validarlos. Se recomienda en proyectos no muy extensos, ya que una Historia de Usuario (HU) no debe poseer demasiada información.

1.5. Herramientas y tecnologías.

La selección acertada de herramientas para emplear en el proceso de desarrollo, apoyado en el dominio que se tenga de las mismas, es crucial para la obtención en tiempo de un producto de calidad. Con esa idea presente, se hizo uso de las herramientas a continuación.

1.5.1. Modelado.

El lenguaje de modelado *UML*[20] en su versión 2.5, permite comunicar la estructura de un sistema complejo, especificar el comportamiento deseado del sistema, comprender mejor lo que se está construyendo y a la vez descubrir oportunidades de simplificación y reutilización. Se basa en el hecho de que un modelo es la simplificación de la realidad (16).

Visual Paradigm en su versión 8.0, se selecciona como herramienta *CASE*[21] de modelado profesional, que utiliza *UML* para la completa representación de las etapas por las que transita un producto de *software*. Este permite la realización de una amplia gama de diagramas como: casos de uso, de actividades, de despliegue, entre otros, así como la generación de código fuente desde los mismos y la documentación asociada al proceso que esté siendo modelado.

1.5.2. Lenguajes de programación.

Lenguaje marcado de hipertexto (*HTML*).

HTML es un lenguaje que se utiliza fundamentalmente en el desarrollo de páginas web. *HTML* son las siglas de *HiperText Markup Language* (Lenguaje de Marcación de Hipertexto) es un lenguaje que se utiliza comúnmente para establecer la estructura y contenido de un sitio web, tanto de texto, objetos e imágenes.

20 Lenguaje de modelado de sistemas de software más conocido y utilizado en la actualidad.

21 Ingeniería de Software Asistida por Computadora (del inglés Computer Aided Software Engineering). Aplicaciones destinadas a aumentar la productividad en el desarrollo de software reduciendo el costo del mismo en términos de tiempo y dinero.

Los archivos desarrollados en *HTML* usan la extensión **.htm o .html**. Funciona por medio de etiquetas que describen la apariencia o función del texto enmarcado (17).

JavaScript.

JavaScript es un lenguaje de programación que se utiliza principalmente para crear páginas web dinámicas. Una página web dinámica es aquella que incorpora efectos como aparición y desaparición de texto, animaciones, acciones que se activan al pulsar botones u otros elementos y ventanas con mensajes de aviso al usuario. Técnicamente, *JavaScript* es un lenguaje de programación interpretado, por lo que no es necesario compilar los programas para ejecutarlos. En otras palabras, los programas escritos con *JavaScript* se pueden probar directamente en cualquier navegador sin necesidad de procesos intermedio (18).

Hojas de estilo en cascada (*CSS*).

Cascading Style Sheets (CSS) conocido en español como Hojas de Estilo en Cascada. Es un mecanismo simple que permite controlar la apariencia de una página web. Describe cómo se muestra un documento en la pantalla, cómo se va a imprimir o incluso cómo va a ser pronunciada la información presente en ese documento a través de un dispositivo de lectura. Esta forma de descripción de estilos le permite a los desarrolladores web controlar el estilo y formato de múltiples páginas web al mismo tiempo. Cualquier cambio en el estilo marcado para un elemento en el *CSS* afectará a todas las páginas vinculadas a ese *CSS* en las que aparezca ese elemento (19).

1.5.3. Framework de desarrollo.

Para el desarrollo de la herramienta se realizó un análisis teniendo presente su objetivo intrínseco: el trabajo con la gestión de los repositorio utilizando *Aptly*. El *framework* de desarrollo que se utilizó fue *Node.js*, ya que proporciona una rica biblioteca de varios módulos *JavaScript* que simplifica el desarrollo de la aplicación web y esta desarrollado tanto para crear código de parte del cliente como del servidor .

Node.js es una plataforma de desarrollo construida en la parte superior de la máquina virtual *JavaScript* de V8 de *Google*. Mientras que los motores de *JavaScript* (incluyendo V8) se ejecutan tradicionalmente en

los navegadores Web para formar el lado cliente de una aplicación cliente / servidor, las bibliotecas *Node.js* se centran en la creación de aplicaciones de servidor en *JavaScript*. También utiliza un modelo de entrada y salida (E/S) sin bloqueo dirigido a eventos que lo hace ligero y eficiente, perfecto para aplicaciones intensivas de datos en tiempo real que se ejecutan en dispositivos distribuidos, además que reduce el reenvasado de la sobrecarga y el empaquetado que utilizan las tecnologías existentes para hablar de un lado a otro de Internet proporcionando un medio ligero para llevar a cabo la misma tarea (20).

1.5.4. Servidor web.

Para el desarrollo de la herramienta se utilizará el servidor web *Express*. *Express* o simplemente *Express*, es un framework de desarrollo para *Node.js*, publicado como software libre y de código abierto. Está diseñado para la creación de aplicaciones web y *APIs*. Es el *framework* de servidor web estándar para *Node.js*.

1.5.5. Sistema gestor de bases de datos.

En el desarrollo de la herramienta se uso *MongoDB* como sistema gestor de base de datos el cual es una base de datos ***NoSQL*** o no relacional además, es un motor de base de datos documental *Open Source* utilizado en aplicaciones web, que se caracteriza al igual que las bases de datos de este tipo por agrupar documentos dentro de una o varias colecciones, dicho de otra forma y para un mejor entendimiento se podría decir que las colecciones representan a una tabla y los documentos representan a las filas o campos de la tabla (21).

Base de Datos *NoSQL.*

Son sistemas de almacenamiento de información que carecen de un modelo entidad-relación con la finalidad de ofrecer velocidad, escalabilidad y rendimiento ya que permite la manipulación de grandes cantidades de información, las bases de datos *NoSQL* (*not only sql*) emplean una arquitectura distribuida permitiendo almacenar los datos de forma redundantemente, al emplear un sistema distribuido ofrece características que hace de las bases de datos *NoSQL* soluciones robustas al momento de procesar grandes cantidades de información, dichas características se detallan en el denominado teorema de *CAP*

(*Consistency, Availability, Partition tolerance* que en español se traduce como Consistencia, Disponibilidad, Tolerancia de las particiones), la consistencia permite mantener la información coherente y consistente ante cualquier cambio u operación que se realizase, la disponibilidad permite mantener siempre la información disponible ante cualquier fallo y la tolerancia de particiones permite mantener funcionando al sistema aunque el sistema se encuentre dividido es decir cualquiera de sus nodos presente caídas o fallos parciales (21).

1.5.6. Entorno de desarrollo.

Webstorm en su versión 2016.3 es un *IDE* de *JavaScript* ligero y potente. Puede manejar fácilmente el complejo desarrollo del lado del cliente y del servidor con *Node.js*. Además de ofrecer soporte para todas las tecnologías que son necesarias para desarrollar una aplicación web. Provee funcionalidades que permiten una experiencia única y aumentan en gran medida la productividad (22):

- **Asistencia de codificación inteligente:** *WebStorm* le ayuda a escribir código mejor gracias a la finalización de código inteligente, detección de errores en la marcha, potente navegación y refactorización.
- **Soporte para las últimas tecnologías:** el *IDE* proporciona soporte de primera clase para *JavaScript, Node.js, HTML y CSS.*
- **Integración perfecta de herramientas:** gracias a la integración con herramientas como *Grunt task runner, linters, npm,* puede minimizar el uso de la línea de comandos. Pero siempre que necesite una **Terminal**, también está disponible como una ventana de herramientas *IDE*.

- **Depuración, rastreo y pruebas:** utiliza un depurador de gran alcance para JavaScript y *Node.js*. Eficientemente rastrea y perfila código con spy-js[22]. Ejecutar pruebas unitarias con Karma[23] o Mocha[24].

1.6. Conclusiones parciales.

A lo largo de este capítulo han sido expuestos los principales puntos de interés relacionados con los conceptos y definiciones asociados a la gestión de repositorios lo que permitió realizar un profundo análisis de las principales características presentes para la creación y gestión de repositorios donde resaltan las versiones dentro del repositorio, la creación y utilización de *rollbacks* y *snapshots* así como la gestión dentro del repositorio. Se realizó un estudio de las herramientas que realizan la gestión de repositorios y se concluyó que la herramienta *Aptly* es la más adecuada para la propuesta de solución.

El estudio del arte realizado permitió identificar la metodología de desarrollo de software y herramientas que se utilizarán en el desarrollo de la solución que se propone, teniendo como resultado las siguientes: como metodología de desarrollo de software AUP-UCI, como lenguaje de programación se utilizó *HTML* para establecer la estructura y contenido de una aplicación web, *CSS* para darle estilo a la aplicación y *JavaScript* para hacer que la aplicación sea dinámica, el *framework* utilizado fue *Node.js* con el servidor web *Express*, el entorno de desarrollo fue *Webstorm* 2016.3, como herramienta *CASE*: *Visual Paradigm* y como sistema gestor de bases de datos *MongoDB*.

22 Es una herramienta para desarrolladores de JavaScript que permite simplemente depurar y rastrear perfil JavaScript que se ejecuta en diferentes plataformas, navegadores y dispositivos. La herramienta rellena las lagunas existentes en las herramientas de desarrollo de los navegadores y aborda tareas comunes de desarrollo desde un ángulo diferente.

23 Se encarga de ejecutar los test de Javascript según se vayan construyendo, de tal forma que ante cualquier fallo el desarrollador se dará cuenta de inmediato.

24 Es un framework de pruebas de JavaScript con múltiples funciones que se ejecuta en Node.js y en el navegador, haciendo que las pruebas asíncronas sean sencillas.

Capítulo 2. Análisis y diseño de solución.

En este capítulo se muestran las fases de **inicio** y **ejecución** de la metodología escogida en el capítulo anterior, además se hace un análisis sobre las características y el diseño que va a tener la propuesta de solución y a su vez validar dichas funcionalidades para comprobar que sean las correctas para el usuario y el funcionamiento de la misma y así no ocurran grandes números de fallas dentro de esta. A continuación se describe la propuesta de solución para darle cumplimiento a la problemática planteada en el documento.

2.1 Propuesta de solución.

Se implementará una herramienta web haciendo uso de la herramienta *Aptly* para la administración de los repositorios de la Distribución Cubana *GNU/Linux* Nova, la cual será capaz de dar solución a los problemas planteados en el capítulo anterior. La aplicación tendrá el nombre de “Gestor de repositorios de Nova”. Se utilizará para la realización de la misma las tecnologías escogidas en el capítulo anterior.

2.2. Especificación de requisitos.

Los requisitos son características requeridas por sistema que posibilitan cumplir con un objetivo o solucionar un problema determinado. Estos son necesarios para el cumplimiento de las especificaciones del cliente.

2.2.1. Requisitos funcionales.

Los requisitos funcionales explican en detalle las tareas que el sistema debe ser capaz de realizar. A continuación, se muestran los requisitos funcionales de la aplicación antes propuesta.

RF 1: gestionar usuario.

RF 2: autenticar usuario.

RF 3: crear fichero de configuración de *Aptly*.

RF 4: modificar fichero de configuración de *Aptly.*

RF 5: crear repositorio.

RF 6: eliminar repositorio.

RF 7: listar repositorios.

RF 8: modificar repositorios.

RF 9: filtrar repositorio.

RF 10: crear repositorios públicos.

RF 11: eliminar repositorios públicos.

RF 12: filtrar repositorios públicos.

RF 13: añadir paquetes en el repositorio.

RF 14: eliminar paquetes en el repositorio.

RF 15: listar paquetes del repositorio.

RF 16: filtrar paquetes del repositorio.

2.2.2. Requisitos no funcionales.

Según *Somerville* los requisitos no funcionales o requerimientos no funcionales no son más que limitaciones sobre servicios o funciones que ofrece el sistema. Incluyen restricciones tanto de temporización y del proceso de desarrollo, como impuestas por los estándares. Los requerimientos no funcionales se suelen aplicar al sistema como un todo, más que a características o a servicios individuales del sistema (23).

Requerimientos de confiabilidad:

RNF 1: el servidor donde estará la herramienta de gestión de repositorios debe contar con un sistema de respaldo eléctrico.

Requerimientos de espacio:

RNF 2: el servidor debe contar como mínimo de un espacio en disco de 1TB.

Requerimientos de rendimiento:

RNF 3: el servidor debe contar como mínimo con un procesador *Intel Core i3-2120 a 2.4GHz* y 4 gb de RAM.

RNF 4: el servidor debe constar con un servidor web con *Express* y una base de datos con *MongoDB*.

RNF 5: la herramienta debe ser escalable, permitiendo incorporarle nuevas funcionalidades sin afectar las existentes.

Requerimientos de Seguridad.

RNF 6: la herramienta debe contar con un control de acceso basado en roles:

Rol administrador: administra los usuarios de la herramienta y los repositorios.

Rol usuario: solamente administra los repositorios.

2.2.3. Validación de requisitos.

La validez de los requisitos es muy importante antes de comenzar un desarrollo de *software* y tiene como objetivo comprobar que estos son correctos. Para ello debe de hacerse una comprobación de la correspondencia entre la descripciones iniciales y si el modelo es capaz de responder al planteamiento inicial. Para llevar a cabo esto, se suele realizar comprobando que el modelo obtenido responde de la misma forma deseada que la que el cliente pide o si otras respuestas del modelo convencen al cliente. En algunos casos será necesario construir prototipos con una funcionalidad similar muy reducida para que el cliente se haga una idea aproximada del resultado. Los parámetros a validar en los requisitos son :

- **Validez:** no basta con preguntar a un usuario, todos los potenciales usuarios pueden tener puntos de vista distintos y necesitar otros requisitos.
- **Consistencia:** no debe haber contradicciones entre unos requisitos y otros.

- **Completitud:** deben estar todos los requisitos. Esto es imposible en un desarrollo iterativo, pero, al menos, deben estar disponibles todos los requisitos de la iteración en curso.
- **Realismo:** se pueden implementar con la tecnología actual.
- **Verificabilidad:** tiene que existir alguna forma de comprobar que cada requisito se cumple.

Se escogió para la validación de los requisitos de la herramienta la técnica prototipo de interfaz de usuario.

2.2.4. Prototipo de interfaz de usuario.

El prototipo de interfaz de usuario es una técnica de representación aproximada de la interfaz de usuario de un sistema *software* que permite a clientes y usuarios entender más fácilmente la propuesta de los ingenieros de requisitos para resolver sus problemas de negocio. Los dos tipos principales de prototipos de interfaz de usuario son:

- **Desechables**: se utilizan sólo para la validación de los requisitos y posteriormente se desechan. Pueden ser prototipos en papel o en *software*.
- **Evolutivos:** una vez utilizados para la validación de los requisitos, se mejora su calidad y se convierten progresivamente en el producto final.

A continuación se muestra la Ilustración 2 con el prototipo de interfaz de usuario del tipo Evolutivo, los demás se encuentran ilustrados en el Anexo 1.

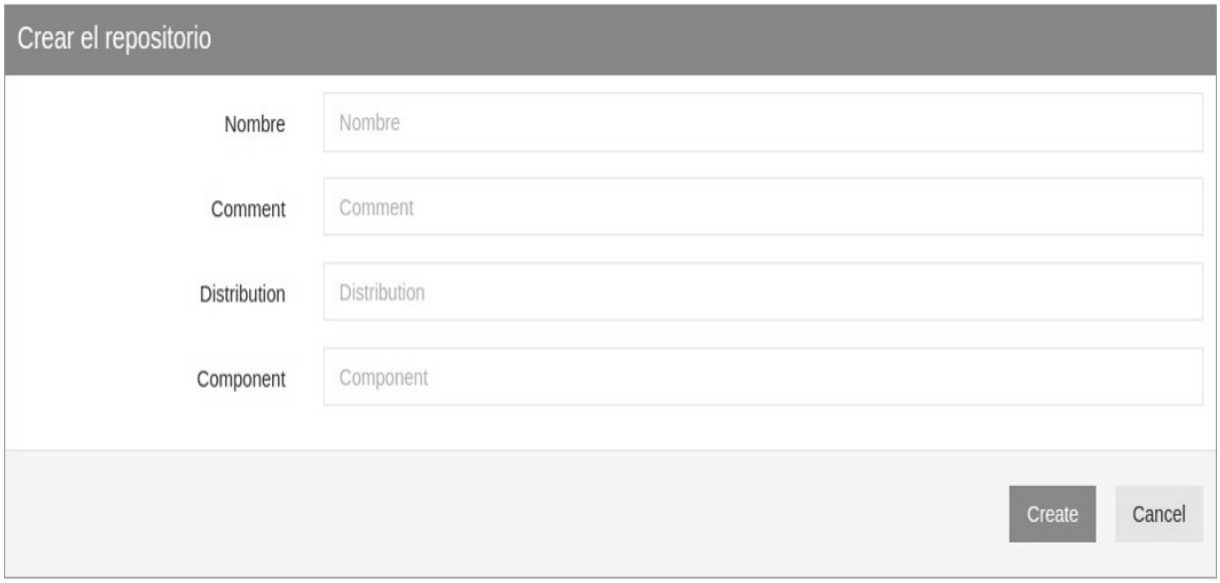

Ilustración 2: Prototipo de interfaz de usuario del tipo Evolutivo: Crear repositorio.

2.4. Historias de usuarios.

Las historias de usuarios conforman la parte central del Escenario 4 de la metodología *AUP-UCI* pues definen lo que se debe construir en el proyecto de *software*. Tienen una prioridad (Alta, Media, Baja) asociada, definida por el cliente y la importancia que tiene para el funcionamiento de la herramienta las funcionalidades antes mencionadas. El riesgo de desarrollo está dado por la ausencia del desarrollador por enfermedad o por pérdida de información imprescindible. El tiempo de cada funcionalidad será estimado por el desarrollador. La estimación del tiempo está dada de acuerdo al tiempo que toma la tarea en ser desarrollada. A continuación, se describen ejemplos de historias de usuario para el desarrollo de la herramienta.

Historia de Usuario	
Número: HU1	**Usuario:** Administradores
Nombre de historia: Autenticar usuario	

Prioridad: Media	**Riesgo en desarrollo:** Ausencia del desarrollador por enfermedad o pérdida de información imprescindible.
Tiempo estimado: 1 semana.	
Programador responsable: Enmanuel C. de la Cruz Mora.	
Descripción: El usuario inserta su identificador (Ejemplo: ecdelacruz) y contraseña en el formulario de autenticación para acceder a las funcionalidades del sistema.	
Observaciones: Si el sistema no identifica al usuario le mostrará una notificación sobre este problema.	

Tabla 2: Historia de Usuario: Autenticar usuario.

Historia de Usuario	
Número: HU2	**Usuario:** Administradores
Nombre de historia: Crear repositorio	
Prioridad: Alta	**Riesgo en desarrollo:** Ausencia del desarrollador por enfermedad o pérdida de información imprescindible.
Tiempo estimado: 2 semana	
Programador Responsable: Enmanuel C. de la Cruz Mora.	
Descripción: El usuario activa el botón “Crear repositorios” para crear el repositorio, proporciona los datos correspondientes para la confección del repositorio.	
Observaciones: El sistema debe notificar al usuario cuando la operación termine o si dejó de llenar algún campo importante .	

Tabla 3: Historia de Usuario: Crear repositorio.

2.5.Arquitectura.

En los últimos años la arquitectura de *software* ha tenido una gran atención en el campo de la ingeniería de *software*. Con el desarrollo de las tecnologías y el aumento en el campo económico, ha quedado en evidencia que un diseño previo y cuidadoso de la arquitectura de un producto puede reducir enormemente el número de fallas del mismo. Indicadores de calidad como la eficiencia, usabilidad, confiabilidad y seguridad pueden ser verificados y estimados con respecto al diseño de la arquitectura escogida, antes de que cualquier código sea escrito.

Por lo antes planteado, se decide seleccionar para el desarrollo de la herramienta la arquitectura ***Model – View – View Model*** (Modelo - Vista - Vista Modelo)**.** Esta arquitectura está basada en la arquitectura Modelo-Vista-Controlador. El Modelo es el responsable de todos los datos de la aplicación y de la lógica de negocios relacionada, la Vista es la encargada de mostrar los datos al usuario y de permitir la manipulación de los datos de la aplicación y la Vista-Modelo es responsable de implementar el comportamiento de la vista para responder a las acciones del usuario y de exponer los datos del modelo de forma tal que sea fácil usar enlaces en la vista.

Model**:** en el modelado se verifica las credenciales del usuario para su autenticación. Una vez que el mismo tenga permisos podrá crear repositorios. También debido a que múltiples usuarios pueden crear un mismo repositorio, una vez que deseen agregarlo a la lista de repositorios, se debe garantizar que no existan repositorios duplicados. Para ello se realiza la comparación de los datos del repositorio a crear, con aquellos que ya fueron introducidos en la listas de repositorios. En caso de existir una coincidencia se le notifica al usuario que ya hay un repositorio similar al suyo. De esta forma se reduce la redundancia al mantener en el servidor un solo repositorio por cada repositorio creado, el cual utilizarían todos los usuarios. Además se crea y se configura el fichero de Aptly para que la herramienta pueda ser usada.

View**:** en el navegador del usuario es descargada una vista de una sola página de *Node.js*, que emula el comportamiento modelo-vista-controlador, a través de la carga dinámica de las vistas que manejan los recursos que el usuario necesite. Estos serían todos los formularios presentes en la herramienta para la interacción con el usuario, ejemplos de estos son: **Configurar fichero**, **Crear repositorio** y **Añadir paquetes.**

View Model**:** en la carpeta ***routers***, en el fichero ***index.js*** se define el comportamiento que tendrán los formularios de la herramienta para responder a las acciones del usuario y se muestran los datos del modelo de forma tal que sea fácil usar enlaces en la vista.

2.6. Diagrama de Paquetes.

Un diagrama de paquetes muestra cómo un sistema está dividido en agrupaciones lógicas y las dependencias entre esas agrupaciones. Dado que normalmente un paquete está pensado como un

directorio, los diagramas de paquetes suministran una descomposición de la jerarquía lógica de un sistema. Además, los paquetes pueden estar anidados uno dentro de otro y unos paquetes pueden depender de otros y se pueden utilizar para plantear la arquitectura de la herramienta a nivel general. El diagrama de paquetes de la herramienta a desarrollar esta compuesto por los tres paquetes principales de la arquitectura escogida, dentro del paquete ***View***, se encuentran los paquetes ***pages*** y ***partials*** además del fichero ***index.ejs*** y dentro de ***partials y pages*** se encuentran los ficheros necesarios para la visualización de la herramienta para la gestión de los repositorios los cuales a su vez se relacionan en el paquete ***View Model*** con el fichero ***index.js*** para definir el funcionamiento de las vistas, también desde el paquete ***pages*** se crean los paquetes ***repository*** y ***db users*** además del fichero de A*ptly* en el paquete ***Model*** los cuales tienen la información de los repositorios, los usuarios creados y el fichero de configuración de *Aptly*, por último el fichero ***index.ejs*** del paquete ***View Model*** antes descrito usa la información almacenada de los paquetes ***repository*** y ***db users*** y la del fichero de configuración de *Aptly* para mostrarla en la vista o comparar la información que se desea entrar en ella con la información que ya se tiene almacenada. En la Ilustración 3 se muestra el diagrama de paquete de la propuesta de solución descrita al inicio de este capítulo.

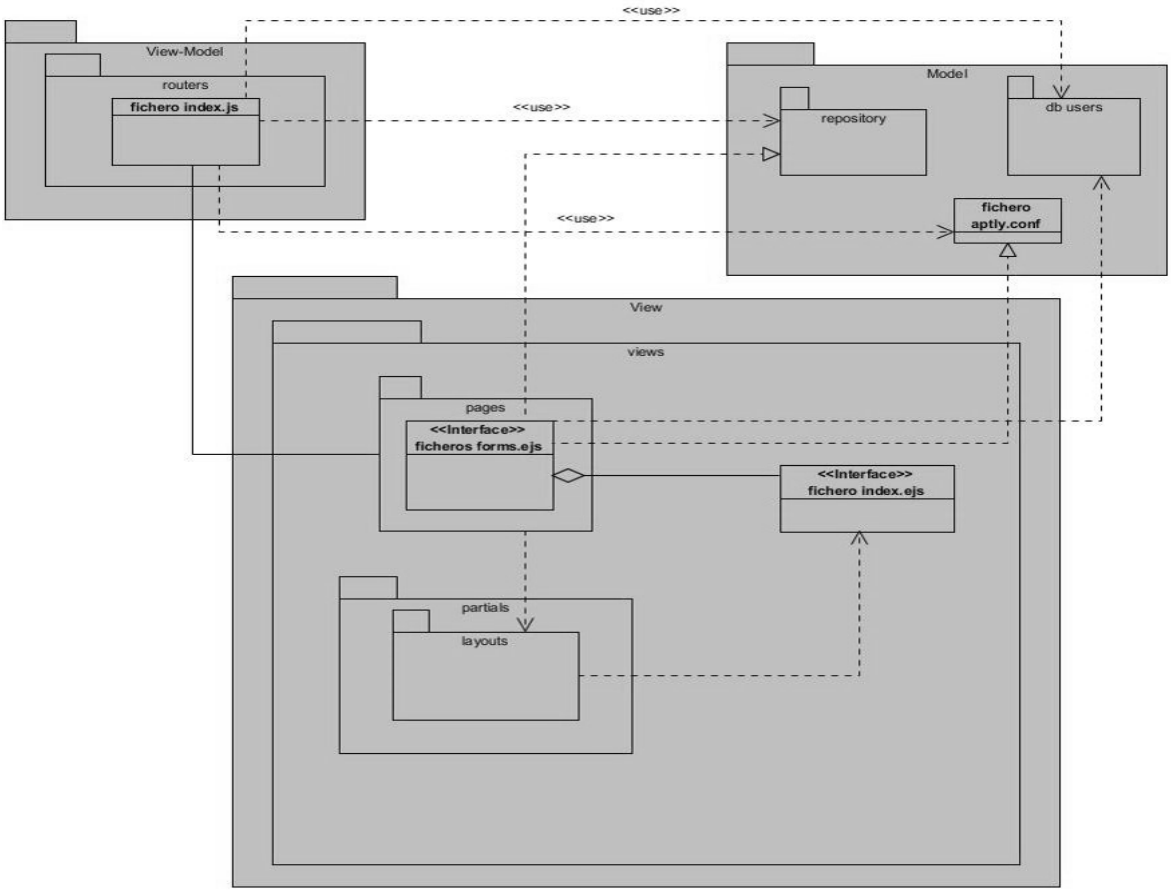

Ilustración 3: Diagrama de paquete.

2.7. Diagrama de Componentes.

Los diagramas de componentes describen los elementos físicos del sistema y sus relaciones. Muestran las opciones de realización incluyendo código fuente, binario y ejecutable. Los componentes representan todos los tipos de elementos software que entran en la fabricación de aplicaciones informáticas. Pueden ser simples archivos o bibliotecas cargadas dinámicamente. Las relaciones de dependencia se utilizan en los diagramas de componentes para indicar que un componente utiliza los servicios ofrecidos por otro

componente. Un diagrama de componentes representa las dependencias entre componentes de *software*, incluyendo componentes de código fuente, componentes del código binario y componentes ejecutables. Un módulo de *software* se puede representar como componente. A continuación se presenta el diagrama de componentes de la herramienta a realizar.

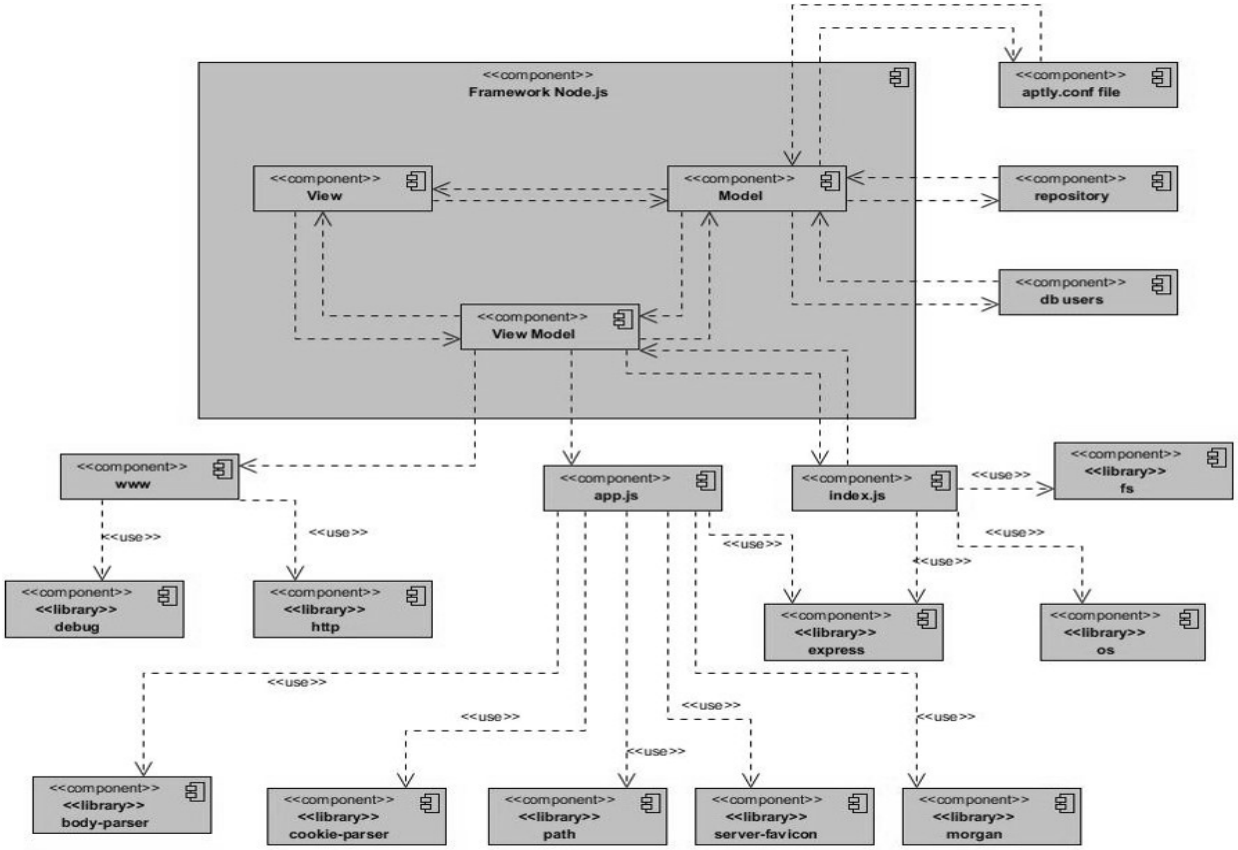

Ilustración 4: Diagrama de componente.

Los componentes de la herramienta *framework Node.js*. Dentro de este componente están los componentes: *View*, *Model* y *View Model*, estos componentes están dados por la arquitectura escogida. El componente *View* se relaciona con los componentes *Model* y *View Model*, los cuales brindan toda la

información y las funcionalidades necesarias para que el usuario pueda interactuar con la herramienta de forma fácil y segura.

En el componente *Model* se encuentran los componentes *aptly.conf file*, *repository* y *db users*. El componente *aptly.conf file* tiene dentro la información necesaria para la configuración de la herramienta *Aptly* la cual se utiliza en la herramienta desarrollada para la administración de los repositorios, además contiene todos los repositorios creados por los usuarios en el componente repository y un componente *db users* con los usuarios autenticados que serán los encargados de administrar los repositorios en la herramienta.

Finalmente dentro del componente ***View Model*** se encuentran los componentes y las librerías necesarias para el correcto funcionamiento de la herramienta ejemplos de estos son: dentro del componente ***index.js*** se encuentra la librería ***fs,*** que es la encargada de la creación del fichero de ***aptly.conf,*** la librería ***express,*** se encuentra dentro de los componentes ***index.js*** y ***app.js*** y se usa para la unión de la herramienta con el servidor web ***Express*** y la librería ***http*** se usa para el enlace de las páginas que tendrá la herramienta a desarrollar.

2.8. Patrones de diseño.

Los patrones de diseño son la base para la búsqueda de soluciones a problemas comunes en el desarrollo de *software* y otros ámbitos referentes al diseño de interacción o interfaces. Un patrón de diseño resulta ser una solución a un problema de diseño. Para que una solución sea considerada un patrón debe poseer ciertas características. Una de ellas es que debe haber comprobado su efectividad resolviendo problemas similares en ocasiones anteriores. Otra es que debe ser reutilizable, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias (24). Para la propuesta de solución se hizo uso de varios Patrones Generales de Software para Asignar Responsabilidad o por sus siglas en inglés (***GRASP***).

El patrón **Bajo Acoplamiento** describe la necesidad de tener una alta reutilización de los componentes con una mínima dependencia entre ellos. En la arquitectura utilizada en la propuesta de solución se evidencia que en la comunicación entre las capas ***Model***, ***View*** y la ***View Model***, se produce una clara

separación de responsabilidades en cada una de ellas. Además, el procesamiento de datos queda encapsulado en cada una independientemente de su representación interna, haciendo posible su reutilización con una mínima dependencia. Un ejemplo que pone en evidencia lo antes planteado es la creación de un repositorio: una vez que el usuario completa el formulario añade el repositorio al servidor, luego el ***View Model*** extrae la información necesaria para mostrarla en las vistas y los envía al ***Model*** donde dicha información puede ser consumida por el usuario a través de las búsquedas de paquetes o de los mismos repositorios, sin tener conocimiento de cómo están representados internamente.

Por otra parte en las vistas se hace uso de las búsquedas. Este proceso fue implementado de forma tal que fuera posible su reutilización en cualquier funcionalidad que requiera o necesite hacer búsquedas ya sea de un repositorio o un paquete en específico. Los repositorios están en un mismo servidor para cualquier usuario que acceda a la herramienta, siendo posible acceder a las funcionalidades que se encargan de su administración desde cualquier parte de la herramienta.

El patrón **Experto** plantea que se debe asignar la responsabilidad a la entidad que posee la información necesaria para cumplir con la responsabilidad requerida. Este patrón se utilizó en el ***View Model*** principalmente en el formulario encargado de la autenticación en el ***View***, ya que este se encarga de encapsular la lógica referente al manejo de usuarios y contraseñas además de la comunicación con el servidor web para ejecutar el mecanismo de autenticación.

Además de los patrones ***GRASP*** existen otros patrones conocidos como la banda de los cuatro o por sus siglas en inglés ***GOF*** (*Gang Of Four*) estos patrones solucionan problemas de creación de instancias y ayudan a encapsular y abstraer dicha creación.

El patrón de **Fachada** viene motivado por la necesidad de estructurar un entorno de programación y reducir su complejidad con la división en subsistemas, minimizando las comunicaciones y dependencias entre estos. Se aplicará el patrón fachada cuando se necesite proporcionar una interfaz simple para un subsistema complejo o cuando se quiera estructurar varios subsistemas en capas, ya que las fachadas serían el punto de entrada a cada nivel o sea, es utilizado cuando es necesario tener una interfaz única para acceder a un conjunto de funcionalidades de un subsistema. El ***View Model*** está construido en forma

de ***api*** y para el acceso a los recursos que él brinda, es necesario realizar una petición *HTTP* (***GET, POST, PUT, DELETE***). Por ejemplo: para crear un repositorio debe hacerse un *POST* al ***api*** de *Aptly* y para modificarlo, se debe hacer una petición *GET* al mismo, ya que con el ***api*** de la herramienta *Aptly* es con lo que se cuenta para mantener la comunicación con la herramienta a ser desarrollada.

2.9. Conclusiones parciales.

En este capítulo se presentó la propuesta de solución para darle respuesta a la problemática planteada al inicio del trabajo, además se identificaron y validaron los requisitos funcionales y no funcionales que fueron posteriormente descritos en las historias de usuarios. La selección de la arquitectura ***Model*** -***View*** -***View Model*** permitió que la herramienta pueda ser utilizada para la gestión de los repositorios. Se realizaron los diagramas de paquetes y de componentes para contribuir a un mejor entendimiento de la herramienta a desarrollar, así como la utilización de los distintos patrones de diseño **Bajo Acoplamiento, Experto** y el patrón de **Fachada** los cuales contribuyeron a facilitar las búsquedas de soluciones a los problemas que se puedan presentar en el desarrollo de la herramienta.

Capítulo 3. Implementación y Pruebas.

Con el objetivo de materializar la solución propuesta de la presente investigación y cumplir con los requisitos obtenidos en el capítulo anterior, se lleva a cabo la fase de implementación y pruebas. Se continúa la fase de **ejecución** y se realiza la fase de **cierre** de la metodología escogida anteriormente, se recogen los estándares de codificación que se emplearán en el desarrollo de la herramienta así como el diagrama de despliegue y finalmente se muestran las pruebas realizadas para la validación del correcto funcionamiento de la propuesta de solución.

3.1. Estándares de codificación.

Entre las buenas prácticas a aplicar durante el proceso de desarrollo de un *software* se encuentran la refactorización del código y la propiedad compartida del mismo, de forma que todo el personal pueda corregir y entender cualquier parte del producto. Un estándar de codificación es necesario pues debe existir un estilo consistente a lo largo de todo el desarrollo de la herramienta y no el preferido por cada programador involucrado en este proceso. Esto posibilita que se pueden añadir nuevas funcionalidades, modificar ya existentes o depurar errores con gran facilidad y obtener un código con mayor legibilidad. A continuación, se define el estándar a utilizar en la implementación de la herramienta a desarrollar:

- La declaración de variables, funciones y constantes se hará en inglés.
- Se empleará el estilo ***lowerCamelCase***[25] para nombres de identificadores, donde todos deberán comenzar con una letra.
- Las constantes serán escritas en minúsculas.

---

25 Estilo de escritura que se aplica a frases o palabras compuestas. La primera letra de cada una de las palabras es mayúscula con la excepción de que la primera letra es minúscula.

- Se hará uso de comentarios en aquellas funcionalidades de mayor complejidad.

3.2. Diagrama de Despliegue.

El diagrama de despliegue es una herramienta muy útil que se emplea para exponer los elementos de configuración del despliegue de una aplicación y las conexiones entre estos elementos. Modelan la arquitectura de un sistema en tiempo de ejecución, permitiendo visualizar los diferentes **nodos**[26] que lo componen así como las relaciones físicas que existen entre ellos. A continuación se muestra el diagrama de despliegue realizado para la herramienta a desarrollar.

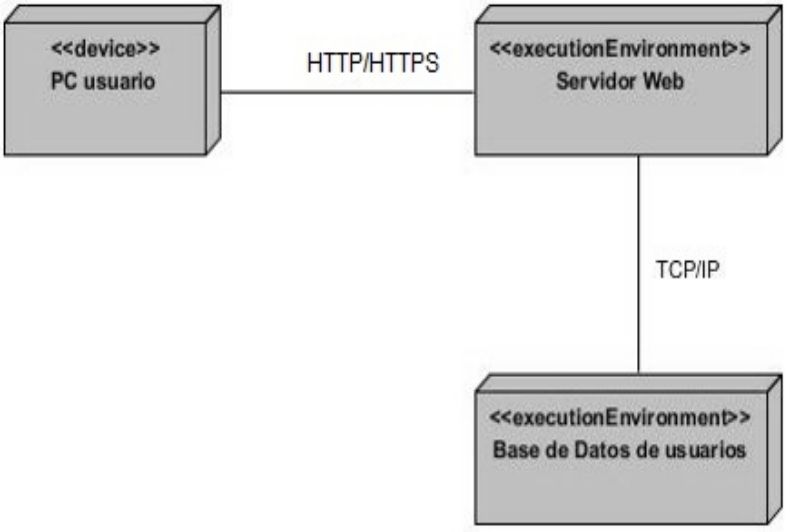

Ilustración 5: Diagrama de Despliegue

La comunicación entre la **PC usuario** y el **Servidor Web**, se realiza utilizando *HTTP* como protocolo para la transferencia de información y desde el **Servidor Web** hacia la **Base de Datos usuarios** la

26 Objeto físico en tiempo de ejecución, es decir, una máquina que se compone habitualmente de, por lo menos, memoria y capacidad de procesamiento.

comunicación es por parte del protocolo *TCP/IP*[27], además la seguridad del sistema se incrementa al ser posible acceder a la **Base de Datos de los usuarios** solamente desde el **Servidor Web**.

PC usuario: Estación de trabajo que necesita un navegador web para conectarse al **Servidor Web** utilizando el protocolo de comunicación *HTTP*.

Servidor Web: Estación de trabajo que hospeda el código fuente de la herramienta desarrollada, así como el fichero de configuración de ***Aptly*** y los repositorios creados por los usuarios.

Base de Datos de usuarios: Base de Datos responsable de almacenar los usuarios del sistema.

3.3. Pruebas de una Webapp.

Las pruebas de una aplicación web o como se conoce en inglés una *Webapp,* es una colección de actividades relacionadas con una sola meta: descubrir errores en el contenido, función, utilidad, navegabilidad, rendimiento, capacidad y seguridad de esa aplicación. Para lograr esto, se aplica una estrategia de pruebas que abarca tanto revisiones como pruebas ejecutables (25).

Estrategia de pruebas de software.

Una estrategia de prueba del *software* integra las técnicas de diseño de casos de prueba en una serie de pasos bien planificados que dan como resultado una correcta construcción del *software*. Y lo que es más importante, una estrategia de prueba del *software* proporciona un mapa a seguir para el responsable del desarrollo del *software*, a la organización de control de calidad y al cliente: un mapa que describe los pasos que hay que llevar a cabo como parte de la prueba, cuándo se deben planificar y realizar esos pasos y cuánto esfuerzo, tiempo y recursos se van a requerir. Por tanto, cualquier estrategia de prueba debe incorporar la planificación de la prueba, el diseño de casos de prueba, la ejecución de las pruebas y la agrupación y evaluación de los datos resultantes (26).

Estrategia de pruebas de software para *Webapps*.

27 Protocolo de Control de Transmisión/Protocolo de Internet (en inglés *Transmission Control Protocol/Internet Protocol*), un sistema de protocolos que hacen posibles servicios *Telnet*, *FTP*, *E-mail*, y otros entre ordenadores que no pertenecen a la misma red.

La estrategia para probar W*ebapps* adopta los principios básicos para todas las pruebas de *software* y aplica una estrategia y tácticas que se usan para sistemas orientados a objetos. Los siguientes pasos resumen el enfoque (25):

1. El modelo de contenido para la *Webapp* se revisa para descubrir errores.

2. El modelo de interfaz se revisa para garantizar que todas las historias de usuarios puedan adecuarse a las necesidades del cliente.

3. El modelo de diseño para la *Webapp* se revisa para descubrir errores de navegación.

4. La interfaz de usuario se prueba para descubrir errores en los mecanismos de presentación y/o navegación.

5. A cada componente funcional se le aplica una prueba de unidad.

6. Se prueba la navegación a lo largo de toda la arquitectura.

7. La *Webapp* se implementa en varias configuraciones ambientales diferentes y se prueba en su compatibilidad con cada configuración.

8. Las pruebas de seguridad se realizan con la intención de explotar vulnerabilidades en la W*ebapp* o dentro de su ambiente.

9. Se realizan pruebas de rendimiento.

10. La W*ebapp* se prueba mediante una población de usuarios finales controlada y monitorizada. Los resultados de su interacción con el sistema se evalúan por errores de contenido y navegación, preocupaciones de facilidad de uso, preocupaciones de compatibilidad, así como confiabilidad y rendimiento de la W*ebapp*.

3.4. Pruebas unitarias.

Las pruebas unitarias enfocan los esfuerzos de verificación en la unidad más pequeña del diseño de *software*: el componente o módulo de *software*. Al usar la descripción del diseño de componente como

guía, las rutas de control importantes se prueban para descubrir errores dentro de la frontera del módulo. La relativa complejidad de las pruebas y los errores que descubren están limitados por el ámbito restringido que se establece para la prueba de unidad (25). Mediante la prueba de la caja blanca el ingeniero del software puede obtener casos de prueba que:

- Garanticen que se ejerciten por lo menos una vez todos los caminos independientes de cada módulo, programa o método.
- Ejerciten todas las decisiones lógicas en las vertientes verdadera y falsa.
- Ejecuten todos los bucles en sus límites operacionales.
- Ejerciten las estructuras internas de datos para asegurar su validez.

Para garantizar la calidad del *software* en cuestión, el desarrollo de la herramienta se llevó a cabo utilizando el método de prueba de caja blanca y una práctica llamada: Desarrollo guiado por pruebas o TDD (del inglés *Test-Driven Development*), que presenta el siguiente ciclo de vida:

1. Confeccionar una prueba para que el código falle y así detectar vulnerabilidades.

2. Escribir el código fuente para que pase dicha prueba.

3. Llevar a cabo la optimización y refactorización del mismo, mientras que la prueba no falle.

4. Repetir este proceso hasta que el proyecto esté completado.

Este flujo de trabajo permite un desarrollo incremental evitando tener que emplear tiempo en encontrar un error en particular, pues el desarrollador es capaz de ir directamente y cambiar la parte de la aplicación que esté presentando problemas. De esta forma es posible saber si la implementación de nuevas funcionalidades o cambios en las ya implementadas están afectando el comportamiento del sistema, solamente con ejecutar dichas pruebas (27).

Método de prueba de caja blanca.

La prueba de caja blanca del *software* se basa en el minucioso examen de los detalles procedimentales. Se comprueban los caminos lógicos del *software* proponiendo casos de prueba que ejerciten conjuntos específicos de condiciones. Se puede examinar el estado del programa en varios puntos para determinar si el estado real coincide con el estado deseado o esperado. La prueba de caja blanca es un método de diseño de casos de prueba que usa la estructura de control del diseño procedimental para obtener los casos de prueba (3).

A continuación se realiza la prueba unitaria con la técnica de camino básico a las funcionalidades **Crear repositorios** la cual creará los repositorios con el nombre y las demás opciones que los usuarios entren en los campos del formulario.

3.4.1. Resultados de las pruebas unitarias.

Luego de numerar las líneas de código, se diseña la gráfica del programa que describe el flujo de control lógico empleando nodos y aristas.

1. **router.post ('/create_repo')**
2. **function (req, res, next)**
3. **aptly.create_repo (req)**
4. **.then(function (response)**
5. **if (response.status === 400)**
6. **return res.render ('./pages/create_repo')**
7. **res.redirect ('/')**
8. **.catch (function (reason)**
9. **res.render ('./pages/create_repo')**

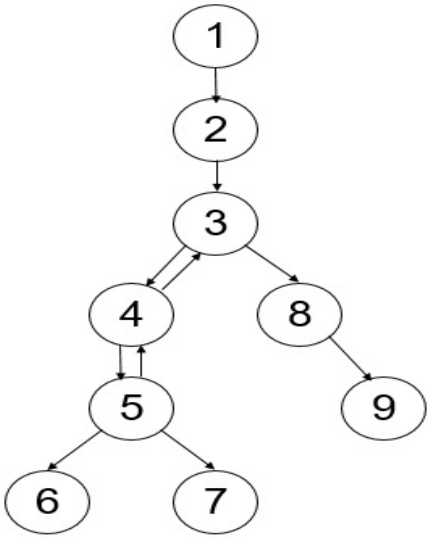

Ilustración 6: Grafo de control lógico del código.

Luego de obtener el grafo de la funcionalidad **Crear repositorios** se calcula la complejidad ciclomática V(G), la cual constituye una métrica de *software* que proporciona una medida cuantitativa de la complejidad lógica del programa (28).

V(G)= (cantidad_aristas – cantidad_nodos) + 2

V(G)= (10 – 9) + 2 = 3

Una ruta independiente es cualquier ruta del programa que ingrese al menos un nuevo conjunto de instrucciones de procesamiento o una nueva condición. La cantidad de rutas independientes establecidas por la complejidad ciclomática antes calculada es de 3, a continuación se presenta una tabla con las rutas y los caminos de ellas.

No. Rutas	Caminos
Ruta 1	1-2-3-4-5-6
Ruta 2	1-2-3-4-5-7
Ruta 3	1-2-3-8-9

Tabla 4: Lista de rutas de la funcionalidad Crear repositorios.

El valor de V(G) ofrece además un límite superior del número de pruebas que debe diseñarse y ejecutarse para garantizar la cobertura de todas las instrucciones. Por este motivo, se diseñan casos de pruebas para ser aplicados a cada ruta independiente de la funcionalidad antes mencionada.

Caso de prueba unitaria	
No. Ruta: Ruta 1	**Ruta:** 1-2-3-4-5-6
Responsable de prueba: Enmanuel C. de la Cruz Mora	
Descripción de la prueba: Finalizar la secuencia en el código que ejecuta el usuario.	
Entrada: Se envía el nombre del repositorio a crear por el usuario, y se finaliza la secuencia del código.	

Resultado esperado: La herramienta muestra la interfaz de Crear repositorios y una notificación.
Evaluación de la prueba: Satisfactoria.

Tabla 5: Caso de prueba unitaria de la primera ruta.

Caso de prueba unitaria	
No. Ruta: Ruta 2	**Ruta:** 1-2-3-4-5-7
Responsable de prueba: Enmanuel C. de la Cruz Mora	
Descripción de la prueba: Finalizar la secuencia en el código que ejecuta el usuario.	
Entrada: Se envía el nombre del repositorio a crear por el usuario, y se finaliza la secuencia del código.	
Resultado esperado: La herramienta muestra la interfaz de inicio y una notificación.	
Evaluación de la prueba: Satisfactoria.	

Tabla 6: Caso de prueba unitaria de la segunda ruta.

Caso de prueba unitaria	
No. Ruta: Ruta 3	**Ruta:** 1-2-3-8-9
Responsable de prueba: Enmanuel C. de la Cruz Mora	
Descripción de la prueba: Finalizar la secuencia en el código que ejecuta el usuario.	
Entrada: Se envía el nombre del repositorio a crear por el usuario, y se finaliza la secuencia del código.	
Resultado esperado: La herramienta muestra la interfaz de Crear repositorios y una notificación.	
Evaluación de la prueba: Satisfactoria.	

Tabla 7: Caso de prueba unitaria de la tercera ruta.

Se realizaron cuatro iteraciones de pruebas unitarias. En la primera iteración se detectaron siete no conformidades, de las cuales se corrigieron cinco; en la segunda, tres no conformidades, dos de la primera iteración y una nueva no conformidad de las cuales se corrigieron dos, en la tercera iteración se corrigió la no conformidad pendiente de la tercera iteración y se realizó una cuarta iteración la cual no arrojo ninguna no conformidad. Las no conformidades detectadas estaban asociadas a errores de

validación y errores de redacción dentro del código. En la gráfica siguiente se visualiza el resultado de dicha prueba.

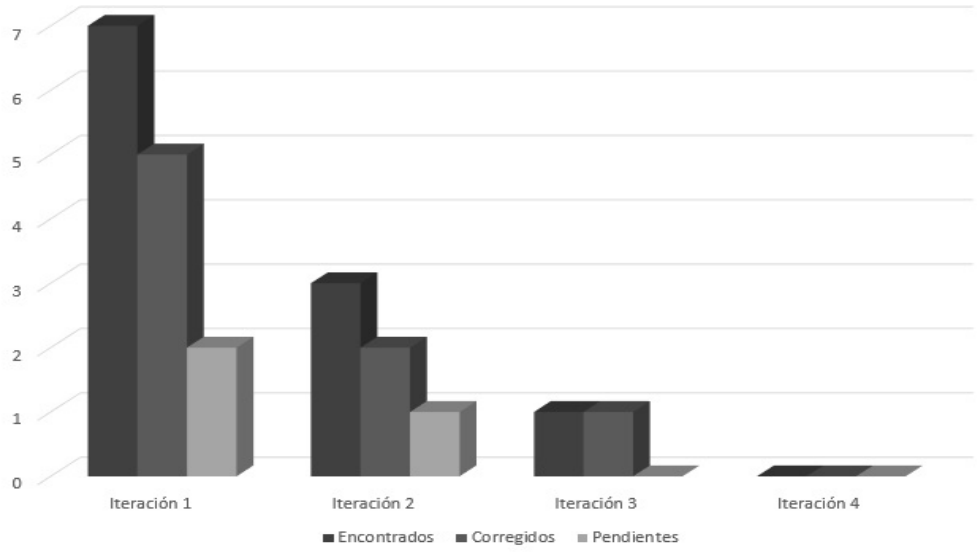

Ilustración 7: Gráfica de los resultados de las pruebas unitarias.

3.5. Pruebas de interfaz de usuario.

La verificación y validación de una interfaz de usuario de *Webapp* ocurre en tres puntos distintos. Durante el análisis de requerimientos, el modelo de interfaz se revisa para garantizar que se da conformidad a los requerimientos de los participantes y a otros elementos del modelo de requerimientos. Durante el diseño, se revisa el modelo de diseño de interfaz para garantizar que se logran los criterios de calidad genéricos establecidos para todas las interfaces de usuario y que los temas de diseño de interfaz específicos de la aplicación se abordaron de manera adecuada. Durante la prueba, la atención se centra en la ejecución de aspectos específicos de la aplicación de la interacción con el usuario, conforme se manifiesten por la

sintaxis y la semántica de la interfaz. Además, la prueba proporciona una valoración final de la usabilidad y se realizan mediante casos de prueba cuyo fin es validar que el *software* cumple con el nivel de calidad requerido (25).

En la herramienta se hace uso del método de prueba de caja negra a la interfaz de dicha herramienta para comprobar y corregir errores que se puedan presentar.

Método de prueba de caja negra.

Las pruebas de caja negra se centran en los requisitos funcionales del *software*, es decir, la prueba de caja negra permite obtener conjuntos de condiciones de entrada que ejerciten completamente todos los requisitos funcionales de un programa. La prueba de caja negra intenta encontrar errores de las siguientes categorías (29):

- Funciones incorrectas o ausentes.
- Errores de interfaz.
- Errores en estructuras de datos.
- Errores de rendimiento.
- Errores de inicialización y de terminación.

Técnica de partición de equivalencia.

La partición de equivalencia es una técnica del método de caja negra que divide el dominio de entrada de un programa en clases de datos de los que pueden derivarse casos de prueba. Un caso de prueba ideal descubre de primera mano una clase de errores (por ejemplo, procesamiento incorrecto de todos los datos carácter) que de otro modo podrían requerir la ejecución de muchos casos de prueba antes de observar el error general (28).

El diseño de casos de prueba para la partición de equivalencia se basa en una evaluación de las clases de equivalencia para una condición de entrada. Con los conceptos introducidos en la sección precedente,

si un conjunto de objetos puede vincularse mediante relaciones que son simétricas, transitivas y reflexivas, se presenta una clase de equivalencia. Una clase de equivalencia representa un conjunto de estados válidos o inválidos para condiciones de entrada (28).

Por lo general, una condición de entrada es un valor numérico específico, un rango de valores, un conjunto de valores relacionados o una condición booleana. Las clases de equivalencia pueden definirse de acuerdo con los siguientes lineamientos (28):

1. Si una condición de entrada especifica un rango, se define una clase de equivalencia válida y dos inválidas.

2. Si una condición de entrada requiere un valor específico, se define una clase de equivalencia válida y dos inválidas.

3. Si una condición de entrada especifica un miembro de un conjunto, se define una clase de equivalencia válida y una inválida.

4. Si una condición de entrada es booleana, se define una clase válida y una inválida.

A diferencia de la prueba de caja blanca, que se lleva a cabo previamente en el proceso de prueba, las pruebas de caja negra tienden a aplicarse durante fases posteriores del proceso de pruebas. A continuación, se muestra el diseño de caso de prueba **Autenticar usuarios** por historias de usuarios de la herramienta.

Caso de prueba de interfaz de usuario	
Número: CDPHU1.	**Nombre:** Autenticar usuarios.
Probador: Enmanuel C. de la Cruz Mora.	
Descripción: Prueba a la funcionalidad Autenticar usuarios.	
Condición de ejecución: El usuario debe estar autenticado.	
Entrada/Pasos de ejecución: 1 El usuario accede a la interfaz de entrada de la herramienta. 2 El usuario llena los campos correo y contraseña del formulario **Acceder**.	

3 El usuario presiona el botón **Acceder** de la interfaz.
4 La herramienta muestra la interfaz principal con el nombre del correo del usuario.

Tabla 8: Caso de prueba de interfaz de usuario: Autenticar usuarios.

Escenarios	Resultados esperados	Evaluación
EC1. El usuario introduce correctamente los datos.	La herramienta autentica al usuario y entra en la interfaz principal mostrando el nombre del correo del usuario.	Satisfactoria.
EC2. El usuario introduce datos incorrectos.	La herramienta no autentica al usuario y muestra el siguiente mensaje: “Datos incorrectos”.	Satisfactoria.
EC3. El usuario deja campos vacíos.	La herramienta no autentica al usuario y muestra los siguientes mensajes: “Correo o contraseña incorrectos”.	Satisfactoria.

Tabla 9: Resultados del Caso de prueba de interfaz de usuario: Autenticar usuarios.

3.5.1. Resultados de las pruebas de interfaz de usuarios.

Para la validación de los requisitos funcionales por medio de las interfaces de los usuarios de la herramienta **Gestión de repositorios de Nova**, se realizaron 3 iteraciones de pruebas de interfaz de usuarios a la misma, en las que se detectaron un total de 25 no conformidades, 20 en la primera iteración de las cuales fueron resueltas 17 quedando pendiente 3 de estas, en la segunda iteración se encontraron 8 no conformidades 3 de la primera iteración y 5 nuevas las cuales fueron todas resueltas y se realizó una tercera iteración donde no se mostró ninguna no conformidad, lo cual arrojo como resultado final que la herramienta funciona correctamente. A continuación se muestran gráficamente estos resultados.

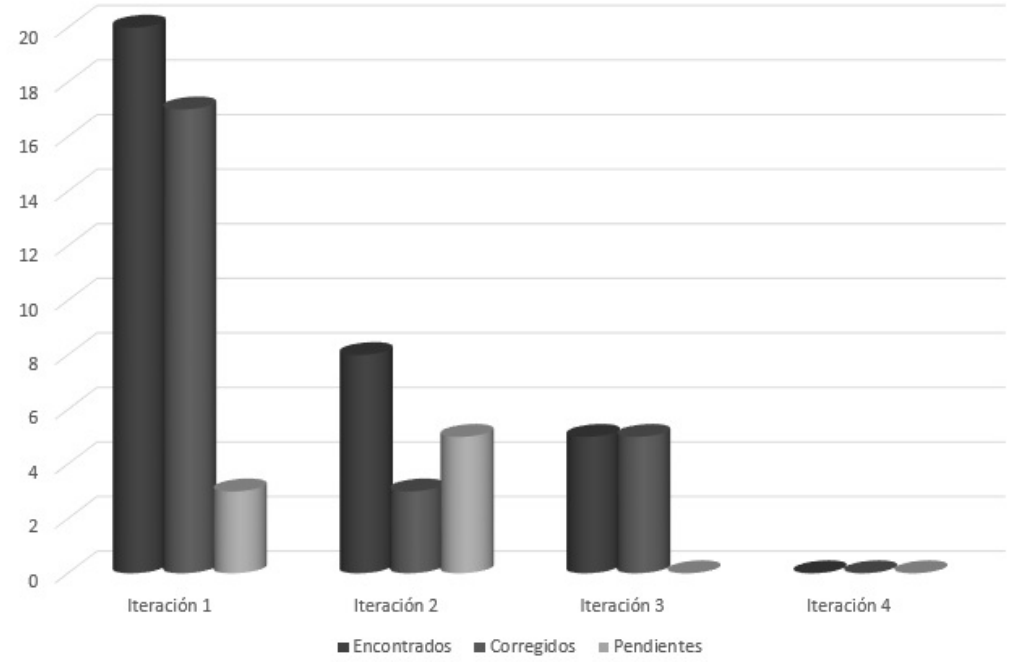

Ilustración 8: Gráfica de los resultados de las pruebas de interfaz de usuario.

3.6. Pruebas de rendimiento.

Las pruebas de rendimiento son un conjunto de pruebas no funcionales que se realizan, para determinar la velocidad de ejecución de una tarea concreta en un sistema bajo condiciones particulares de trabajo.

Los objetivos de estas pruebas son:

- Validar y verificar atributos de la calidad del sistema: uso de los recursos, escalabilidad y fiabilidad.
- Comparación de sistemas para encontrar cuál de ellos funciona mejor.
- Determinar qué componentes del sistema provocan que el conjunto presente rendimientos bajos.

Tipos de Pruebas de Rendimiento

1 Prueba de Carga: prueba de rendimiento que se realiza para observar el comportamiento de una aplicación bajo una cantidad de peticiones esperada.

Objetivos:

- Mostrar los tiempos de respuesta de todas las transacciones importantes.
- Localizar los 'cuellos de botella' de una aplicación.

2 Pruebas de Estrés: Prueba de rendimiento que se realiza para observar el comportamiento de una aplicación bajo una cantidad de peticiones extrema.

Objetivos:

- "**Romper**" la aplicación.
- Determinar cómo rendirá la aplicación si la carga real supera a la carga esperada.

3.6.1. Resultados de las pruebas de rendimiento.

La herramienta desarrollada fue sometida a varias pruebas de rendimiento para medir su funcionamiento. Estas pruebas estuvieron enfocadas en el análisis del comportamiento de los tiempos de respuesta de la funcionalidad: añadir paquetes. Para ello se emplearon dos técnicas diferentes de subida de los paquetes, para comprobar su impacto en dichos tiempos de respuesta:

Añadir paquetes desde el servidor: se envían desde el servidor los paquetes seleccionados para ser subidos al repositorio elegido.

Añadir paquetes desde una pc cliente: se envían desde una pc cliente hacia el servidor los paquetes seleccionados para ser subidos al repositorio elegido.

Las pruebas se llevaron a cabo en un ordenador con un microprocesador *Intel Core i3-2120 a 2.4GHz* y 4 gb de RAM. Para probar el tiempo que la herramienta empleaba en subir un paquete, hasta que se añade en el repositorio escogido, se realizó la carga de 10000 paquetes con extensiones .deb y .udeb con diferentes tamaños. A continuación, se muestran las gráficas comparativas del tiempo de subida de los paquetes desde el servidor y desde una pc cliente.

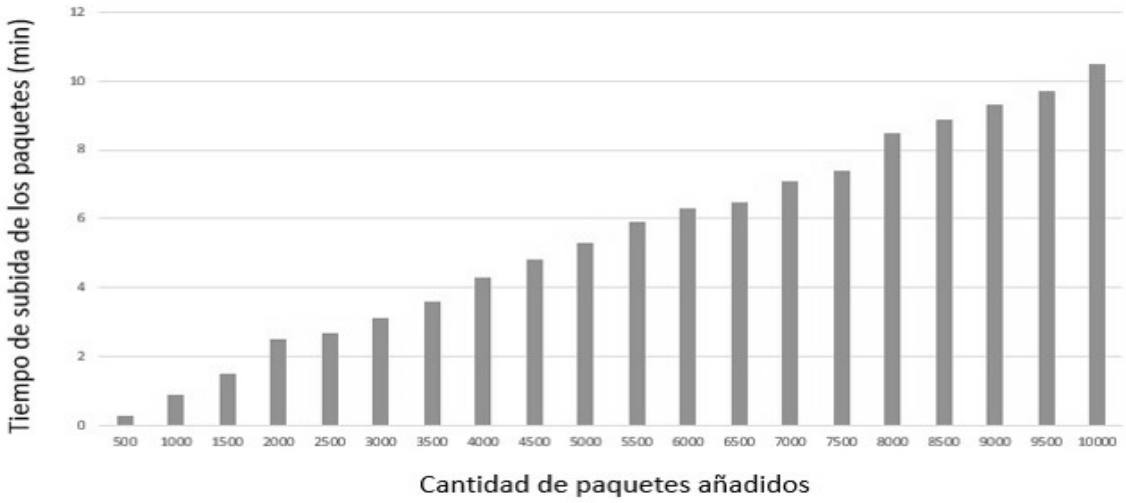

Ilustración 9: Gráfica del resultado del tiempo de subida de los paquetes desde el servidor.

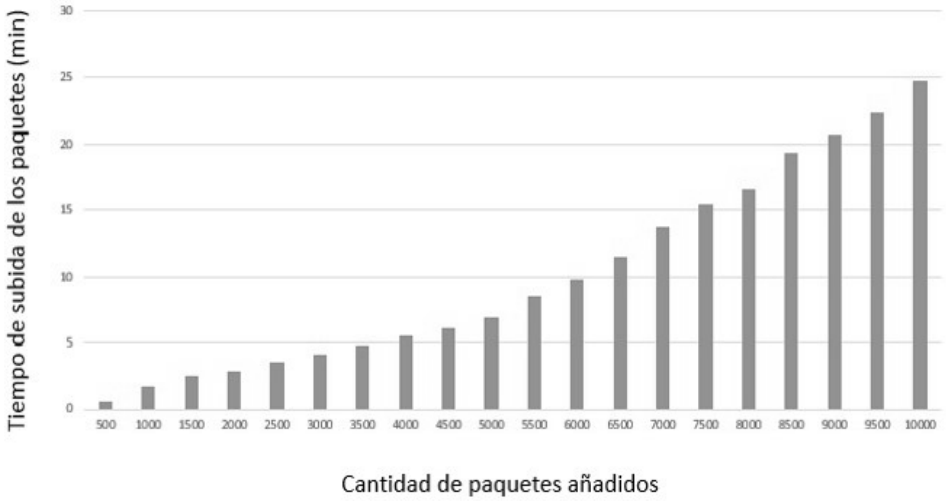

Ilustración 10: Gráfica del resultado del tiempo de subida de los paquetes desde una pc cliente.

A continuación se muestra una tabla con los resultados más importantes que se obtuvieron de las pruebas de rendimiento realizadas a las dos variantes de subida de los paquetes hacia los repositorios.

Tiempo de paquetes	Servidor	PC cliente
Mayor tiempo de subida de los paquetes (min)	10.5	24.8
Menor tiempo de subida de los paquetes (min)	0.3	0.6
Tiempo promedio de subida de un paquete (min)	0.6	1.4

Tabla 10: Resultados del tiempo de subida de un paquete desde una pc cliente y el servidor.

Con la realización de las pruebas de rendimiento se mostró que el tiempo de subida de los paquetes con la herramienta de administración de los repositorios se redujo en un 15% desde una pc cliente y en un 30% desde el servidor. Además, no se produjo pérdida de ningún paquete ni hubo errores a la hora de subida de los mismos hacia el repositorio seleccionado por el usuario.

3.7. Pruebas de aceptación.

Las pruebas de aceptación representan aquella fase del ciclo de vida de desarrollo de *software* en que el equipo de desarrollo y los usuarios tienen que garantizar que el sistema desarrollado se corresponda con los requerimientos definidos en la fase de análisis. Es fundamental que la calidad tanto del código como de la documentación aportada al usuario sea alta y se corresponda con los parámetros adecuados para el *software* que concretamente se esté desarrollando (30).

Las pruebas de aceptación se realizan a partir de las historias de usuarios, donde cada una se convierte en un caso de prueba en el cual el cliente especifica los puntos a probar. Además, una historia de usuario pudiera tener más de una prueba de aceptación, pues se deben realizar las necesarias para garantizar la

calidad del producto final. A continuación, se muestra un caso de prueba de ejemplo para la funcionalidad **Crear repositorio**.

Caso de prueba de aceptación	
Código: HU2CP2.	**Historia de usuario:** Crear repositorio.
Responsable de prueba: Enmanuel C. de la Cruz Mora.	
Descripción: Prueba para comprobar el proceso de creación del repositorio.	
Condiciones de ejecución: El usuario tiene que estar autenticado.	
Entrada/Pasos de ejecución: 1 El usuario accede a la interfaz principal de la herramienta. 2 El usuario presiona la acción **Repositorios** del menú de navegación. 3 El usuario selecciona la funcionalidad **Administrar repositorios** dentro de **Repositorios** en el menú de navegación. 4 La herramienta muestra un formulario con los siguientes datos: • **Nombre.** • **Comentario.** • **Distribución.** • **Componente.** 5 El usuario entra los datos y los envía. 6 La herramienta valida los datos, crea el repositorio y muestra la interfaz principal con una notificación al usuario.	
Resultado esperado: El repositorio es creado por el usuario.	
Evaluación de la prueba: Prueba satisfactoria.	

Tabla 11: Caso de prueba de aceptación: Crear repositorio.

Una vez terminado este proceso el cliente quedó satisfecho, pues todos los casos de prueba se ejecutaron de forma satisfactoria.

3.8. Conclusiones parciales.

En el desarrollo del presente capítulo se realizó el diagrama de despliegue lo cual permitió conocer la distribución física del sistema sobre una arquitectura de *hardware*. Se especificó el uso de los estándares de codificación para lograr obtener claridad y organización en el código fuente de la solución. Se realizaron pruebas unitarias, de interfaz de usuario y de aceptación a la herramienta para comprobar tanto el funcionamiento del código como la interfaz de la misma y los errores encontrados fueron tratados y corregidos en su totalidad.

Conclusiones.

Con el desarrollo de una herramienta para la gestión de los repositorios de la Distribución Cubana de *GNU/Linux* Nova se da cumplimiento al objetivo general planteado. Para llegar a este resultado se concluye lo siguiente:

- Se realizó un estudio de las principales herramientas que realizan la gestión de repositorios en los sistemas *GNU/Linux* con el cual se pudo identificar la herramienta más adecuada para darle solución al problema de la investigación.
- Se realizó el análisis, diseño e implementación de la herramienta para la gestión de los repositorios de la Distribución Cubana de *GNU/Linux* Nova utilizando la herramienta *Aptly*, obteniendo así una herramienta que cumple con las necesidades del cliente y elimina las vulnerabilidades antes existentes.
- Se diseñaron y se realizaron las pruebas de *software*, permitiendo la correcta comprobación del funcionamiento de la solución, dando la posibilidad de realizar el proceso de entrega del *software* a consideración del cliente.

Recomendaciones.

Con la realización del presente trabajo se desarrolló la herramienta para la administración de los repositorios de la Distribución Cubana de *GNU/Linux* Nova. Para futuras investigaciones y desarrollo de la herramienta se recomienda:

- Realizar la configuración de la herramienta “Gestor de repositorios de Nova” con el servidor web Apache para la visualización de los repositorios que se encuentran accesibles para los usuarios.

Referencias Bibliográficas.

1. Quienes usan Linux ? [en línea]. [Accedido 8 de Diciembre 2016]. Disponible desde: http://www.informatica-hoy.com.ar/software-libre-gnu/Quienes-usan-Linux.php

2. PERÉZ, GASTON, GARCÍA, G AND N, IRMA. *Metodología de la Investigación Educacional.* La Habana : Pueblo y Educación, 1996.

3. Centro de Recursos para la Escritura Académica del Tecnológico de Monterrey. [en línea]. [Accedido 8 de Diciembre 2016]. Disponible desde: http://sitios.ruv.itesm.mx/portales/crea/buscar/que/6_lospasos.htm

4. PoliScience » Definición y tipos. [en línea]. [Accedido 17 deMayo 2017]. Disponible desde: http://poliscience.blogs.upv.es/open-access/repositorios/definicion-y-tipos/

5. Definicion de Paquete de software. [en línea]. [Accedido 17 de Mayo 2017]. Disponible desde: http://www.alegsa.com.ar/Dic/paquete_de_software.php

6. ¿Qué es un gestor de paquetes? [en línea]. [Accedido 17 de Mayo 2017]. Available from: https://www.debian.org/doc/manuals/aptitude/pr01s02.es.html

7. Definición de API - Qué es, Significado y Concepto. [en línea]. [Accedido 17 de Mayo 2017]. Disponible desde: http://definicion.de/api/

8. Packaging/PPA - Launchpad Help. [en línea]. [Accedido 25 de Febrero 2017]. Disponible desde: https://help.launchpad.net/Packaging/PPA

9. ¿Qué significa PPA en entornos Linux (Ubuntu / Linux Mint)? [en línea]. [Accedido 25 de Febrero 2017]. Disponible desde: http://vivaelsoftwarelibre.com/ppa-en-linux-que-es/

10. reprepro manual. [en línea]. [Accedido 8 de Diciembre 2016]. Disponible desde: http://www.red-bean.com/doc/reprepro/manual.html

11. REPREPRO. [en línea]. [Accedido 8 de Diciembre 2016]. Disponible desde: https://mirrorer.alioth.debian.org/reprepro.1.html

12. aptly - Debian repository management tool. [en línea]. [Accedido 17 de Mayo 2017]. Disponible desde: https://www.aptly.info/

13. *Choosing An Appropriate System Development Methodology - SelectingDevelopmentApproach.pdf* [en línea]. [Accedido 23 de Febrero 2017]. Disponible desde: https://www.cms.gov/Research-Statistics-Data-and-Systems/CMS-Information-Technology/XLC/Downloads/SelectingDevelopmentApproach.pdf

14. FIGUEROA-DÍAZ, Roberth G. Metodologías tradicionales vs. Metodologías ágiles. *Universidad Técnica Particular de Loja, Escuela de Ciencias en Computación.* [en línea]. 2008. Disponible desde: http://adonisnet. files. wordpress. com/2008/06/articulo-metodologia-de-sw-formato. doc

15. TAMARA RODRÍGUEZ SÁNCHEZ. *Metodología de desarrollo para la Actividad productiva de la UCI.* 3 June 2015. Versión: 1.2.

16. *el-lenguaje-unificado-de-modelado-manual-de-referencia.pdf* [en línea]. [Accedido 24 deFebrero 2017]. Disponible desde: https://ingenieriasoftware2011.files.wordpress.com/2011/07/el-lenguaje-unificado-de-modelado-manual-de-referencia.pdf

17. Lenguaje HTML - Documentación sobre desarrollo web. [en línea]. [Accedido 1 de Marzo 2017]. Disponible desde: https://lenguajehtml.com/

18. EGUILUZ, Javier. *Introducción a JavaScript*. 2009. página 7

19. Guía Breve de CSS. [en línea]. [Accedido 1 de Marzo 2017]. Disponible desde: http://www.w3c.es/Divulgacion/GuiasBreves/HojasEstilo

20. What is Node.js? - Definition from Techopedia. [en línea]. [Accedido 24 de Febrero 2017]. Disponible desde: https://www.techopedia.com/definition/27927/nodejs

21. *UPS - ST002104.pdf* [en línea]. [Accedido 9 de Mayo 2017]. Disponible desde: http://www.dspace.ups.edu.ec/bitstream/123456789/12424/1/UPS%20-%20ST002104.pdf

22. WebStorm: The Smartest JavaScript IDE. [en línea]. [Accedido 24 de Febrero 2017]. Disponible desde: https://www.jetbrains.com/webstorm/

23. Ingenieria-de-Software-Ian-Somerville-9-edicion-español[1] | Jamie Patrick - Academia.edu. [en línea]. [Accedido 30 de Mayo 2017]. Disponible desde: http://www.academia.edu/15366832/Ingenieria-de-Software-Ian-Somerville-9-edicion-espa%C3%B1ol_1_

24. RICARDO BOTERO TABARES. Patrones Grasp y Anti-Patrones: un Enfoque Orientado a Objetos desde Lógica de Programación. *Entre ciencia e ingeniería*. 2011. P. 161 a 173. El presente artículo plantea una estrategia didáctica para introducir el uso de los patrones GRASP (General Responsibility AssignmentSoftware Patterns).

25. *Ingenieria del Software. Un Enfoque Practico - ld-Ingenieria.de.software.enfoque.practico.7ed.Pressman.PDF* [en línea]. [Accedido 9 de Mayo 2017]. Disponible desde: http://cotana.informatica.edu.bo/downloads/ld-Ingenieria.de.software.enfoque.practico.7ed.Pressman.PDF

26. ESTRATEGIAS DE PRUEBA DEL SOFTWARE. [en línea]. [Accedido 9 de Mayo 2017]. Disponible desde: http://www.angelfire.com/my/jimena/ingsoft/guia10.html

27. HARVEY, K. *Test-driven development with Django.* 2015.

28. PRESSMAN, ROGER. *Ingeniería de Software: Un Enfoque Práctico*. Sexta Edición. Nueva York : McGraw-Hill / Interamericana de México, 2005.

29. *Capitulo 4: Pruebas Funcionales y de Campo - capitulo4.pdf* [en línea]. [Accedido 9 deMayo 2017]. Disponible desde: http://catarina.udlap.mx/u_dl_a/tales/documentos/lis/fuentes_k_jf/capitulo4.pdf

30. Pruebas de aceptación orientadas al usuario: contexto ágil para un proyecto de gestión documental - Documat. [en línea]. [Accedido 28 de Mayo 2017]. Disponible desde: https://documat.unirioja.es/servlet/articulo?codigo=5413986

Anexos.

Anexo 1. Prototipo de interfaz de usuario.

Ilustración 11: Prototipo de interfaz de usuario: Registrar usuario.

Ilustración 12: Prototipo de interfaz de usuario: Autenticar usuario.

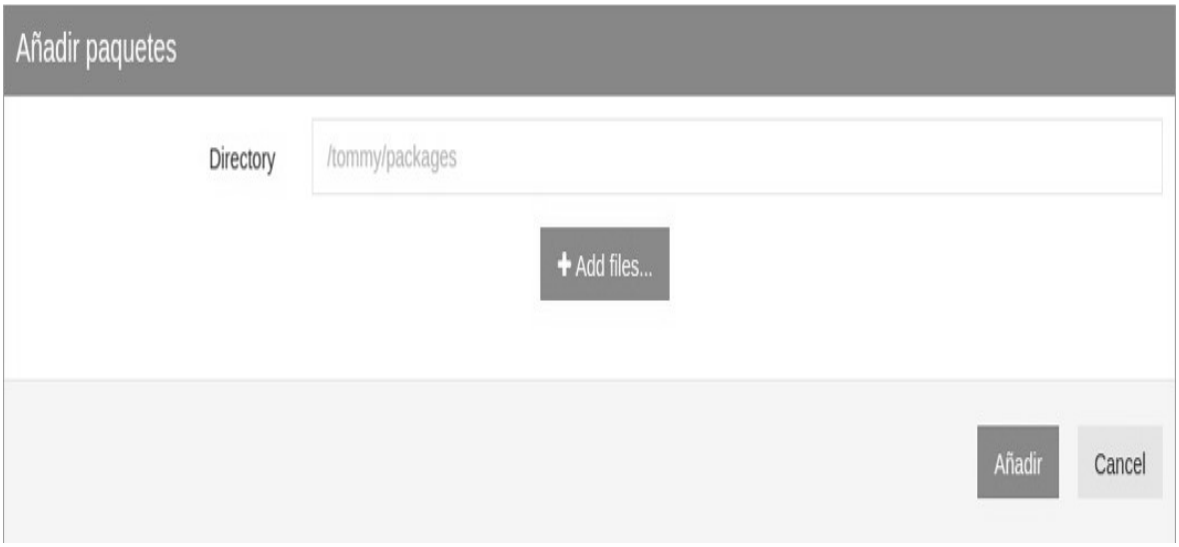

Ilustración 13: Prototipo de interfaz de usuario: Añadir paquetes.

Anexo 2. Historias de usuarios.

Historia de Usuario	
Número: HU3	**Usuario:** Administradores
Nombre de historia: Modificar repositorio	
Prioridad: Media	**Riesgo en desarrollo:** Ausencia del desarrollador por enfermedad o pérdida de información imprescindible.
Tiempo estimado: 1 semana	
Programador Responsable: Enmanuel C. de la Cruz Mora.	
Descripción: El usuario activa el botón ”Modificar” para modificar el repositorio deseado y este a su vez le muestra una vista al usuario que proporciona los datos correspondientes para la modificación del mismo.	
Observaciones: La herramienta debe notificar al usuario cuando la operación termine o si un campo importante no está correcto o se encuentra vacío.	

Tabla 12: Historia de usuario: Modificar repositorio.

Historia de Usuario	
Número: HU4	**Usuario:** Administradores
Nombre de historia: Eliminar repositorio	
Prioridad: Media	**Riesgo en desarrollo:** Ausencia del desarrollador por enfermedad o pérdida de información imprescindible.
Tiempo estimado: 1 semana	
Programador Responsable: Enmanuel C. de la Cruz Mora.	
Descripción: El usuario activa el botón ”Eliminar” para eliminar el repositorio deseado y la herramienta muestra un cartel para pedirle al usuario si en verdad desea eliminar ese repositorio.	
Observaciones: La herramienta muestra la lista de repositorios mostrándole al usuario que el repositorio ya no existe.	

Tabla 13: Historia de usuario: Eliminar repositorio.

Historia de Usuario	
Número: HU5	**Usuario:** Administradores
Nombre de historia: Publicar repositorio	
Prioridad: Media	**Riesgo en desarrollo:** Ausencia del desarrollador por enfermedad o pérdida de información imprescindible.
Tiempo estimado: 1semana	
Programador Responsable: Enmanuel C. de la Cruz Mora.	
Descripción: El usuario activa el botón ”Publicar” que publica el repositorio deseado y la herramienta muestra un cartel para pedirle al usuario si en verdad desea publicar ese repositorio.	
Observaciones: La herramienta muestra la lista de repositorios públicos mostrándole al usuario que el repositorio ya se encuentra público.	

Tabla 14: Historia de usuario: Publicar repositorio.

Historia de Usuario	
Número: HU6	**Usuario:** Administradores
Nombre de historia: Añadir paquetes	
Prioridad: Media	**Riesgo en desarrollo:** Ausencia del desarrollador por enfermedad o pérdida de información imprescindible.
Tiempo estimado: 2 semana	
Programador Responsable: Enmanuel C. de la Cruz Mora.	
Descripción: El usuario activa en el menú de navegación la opción “Añadir paquete” la cual añadirá los paquetes escogidos en el repositorio seleccionado y presiona el botón “Enviar” para terminar la acción.	
Observaciones: La herramienta muestra la lista de paquetes añadidos al repositorio seleccionado en una tabla que presenta la cantidad de paquetes que se pueden observar en cada vista de la misma.	

Tabla 15: Historia de usuario: Añadir paquetes.

Historia de Usuario	
Número: HU7	**Usuario:** Administradores
Nombre de historia: Eliminar paquete	
Prioridad: Media	**Riesgo en desarrollo:** Ausencia del desarrollador por enfermedad o pérdida de información imprescindible.
Tiempo estimado: 1 semana	
Programador Responsable: Enmanuel C. de la Cruz Mora.	
Descripción: El usuario activa el botón ”Eliminar” para eliminar el paquete deseado y la herramienta muestra un cartel para pedirle al usuario si en verdad desea eliminar ese paquete.	
Observaciones: La herramienta muestra la lista de paquetes del repositorio mostrándole al usuario que el paquete ya no existe.	

Tabla 16: Historia de usuario: Eliminar paquete.

Historia de Usuario	
Número: HU8	**Usuario:** Administradores
Nombre de historia: Eliminar repositorios públicos	
Prioridad: Media	**Riesgo en desarrollo:** Ausencia del desarrollador por enfermedad o pérdida de información imprescindible.
Tiempo estimado: 1 semana	
Programador Responsable: Enmanuel C. de la Cruz Mora.	
Descripción: El usuario activa el botón ”Eliminar” para eliminar el repositorio público deseado y la herramienta muestra un cartel para pedirle al usuario si en verdad desea eliminar ese repositorio.	
Observaciones: La herramienta muestra la lista de repositorio públicos mostrándole al usuario que el repositorio ya no existe.	

Tabla 17: Historia de usuario: Eliminar repositorios públicos.