Herramienta de pruebas de integridad al repositorio de Nova

Trabajo de Diploma para optar por el título de Ingeniero

en Ciencias Informáticas

Autor: Pedro Javier Hernández Melgarejo

Tutores: MSc. Juan Manuel Fuentes Rodríguez

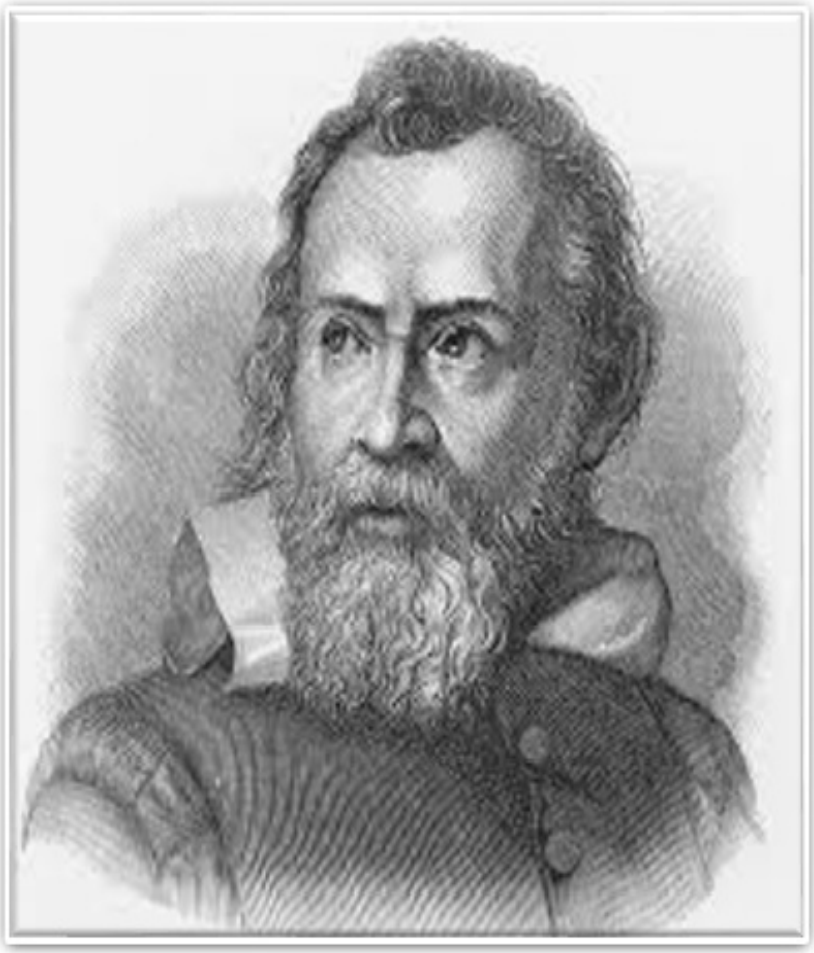
Ing. Yileni Hechavarría González

Universidad de las Ciencias Informáticas.

La Habana, Cuba, junio 2019.

"En cuestiones de ciencia, la autoridad de mi persona no vale el humilde razonamiento de un solo individuo"

Galileo Galilei



Declaración de autoría

Declaro ser el único autor de este trabajo y autorizo a la Universidad de las Ciencias Informáticas a hacer uso del mismo en su beneficio.

Pedro Javier Hernández Melgarejo

_____

Firma del autor

MSc. Juan Manuel Fuentes Rodríguez

_____

Firma del tutor

Ing. Yileni Hechavarría González

_____

Firma del tutor

Dedicatoria

A mis padres Tania Calixta Melgarejo Oviedo y Pedro Alipio Hernández Sánchez por haber hecho tanto por mí en estos más de 24 años, a mi madre por sus sacrificios y a mi padre por estar siempre ahí. A mi hermano Yasel Hernández Melgarejo por ser ejemplo de superación, estudiante y persona. A mi mujer por darme esa beba hermosa, un motivo para luchar mientras tenga vida, por la paciencia y la comprensión. A mis abuelos y padres, sin ellos no hubiese terminado mi carrera. A todos mis familiares, en especial a Nelson y Adriana, por las charlas, preocupaciones y el apoyo incondicional. También a los familiares y amigos que por desgracia no están en vida. A Denis, Alex y Luis Manuel, por ser mis amigos en las buenas y en las malas. Todos ustedes son mi inspiración, parte indispensable de mis éxitos, por cada beso, abrazo y lágrima, por todo el amor y el cariño. A mis compañeros de aula, a los piquetes de fútbol, al resto de mis amistades dentro y fuera de la UCI, son muchos y es imposible mencionarlos a todos pero sepan que esto va por ustedes también.

Agradecimientos

Agradecimientos especiales a mis tutores Yilenis y Juan por por dar su apoyo y empeño para la correcta confección de la tesis. A Denis, Alex y Luis Manuel por ser el empujón y la mano que recibí cuando necesite. A los profesores de la UCI que formaron parte en el largo de mi trayectoria como estudiante. A los ingenieros de CESOL, a mis amistades y familiares. A todas las personas que un modo u otro me han ayudado a la realización de esta tesis. También a los que se me olvidaron.

Resumen

Los repositorios de *software* componen un factor esencial en el uso y desarrollo de las distribuciones de *software* libre. La integridad de paquetes es una característica general descrita por el buen funcionamiento y completitud de los datos; por tanto resulta un buen medidor de calidad. Como contenedor de cientos de paquetes de programas, al repositorio de la distribución cubana GNU/Linux Nova, se le hace necesaria la verificación de la integridad en cada uno de ellos. Este proceso se realiza de forma manual lo que requiere de mucho consumo de tiempo y en ocasiones sea difícil de ejecutar, por lo que el procedimiento es tedioso para los especialistas. La realización del trabajo “Herramienta de pruebas de integridad al repositorio de Nova”, permite analizar las dependencias de los paquetes, así como detectar la presencia de paquetes averiados. Para la elaboración de la herramienta de pruebas de integridad se emplearon Bash y Python como lenguajes de programación, Visual Studio Code como entorno de desarrollo integrado y Visual Paradigm como herramienta de modelado. Una vez terminado el desarrollo de la herramienta, se realizaron evaluaciones mediante la aplicación de pruebas. Este *software* da solución al descubrimiento de cambios o modificaciones en los paquetes del repositorio y agiliza el proceso de búsqueda de posibles errores.

Palabras clave: integridad, métodos de pruebas de integridad, nova, paquete de *software*, repositorio.

Índice

Índice de Figuras

Índice de tablas

Introducción

Un repositorio de *software* libre es un contenedor de paquetes de programas y actualizaciones preparados para una distribución determinada que permite a los usuarios el acceso a estos programas y el uso de los mismos. Estos repositorios por lo general se encuentran alojados en un servidor de la red, lo que permite hacer múltiples peticiones de manera simultánea. Por tanto, la estructura de la red debe ser eficiente para cumplir con las demandas de acciones sobre el repositorio y mantener a disponibilidad dichos paquetes. Generalmente cada distribución tiene un repositorio correspondiente donde los usuarios pueden auxiliarse de los servicios [1].

Uno de los puntos esenciales en la informatización de la sociedad cubana lo constituye el proceso de migración a *software* libre. La Universidad de las Ciencias Informáticas (UCI) como centro de estudio dedicado a la docencia, investigación y producción de *software* para la sociedad cubana y el mundo, posee un conjunto de proyectos de los cuales varios están encaminados al desarrollo de *software* libre; en especial a la distribución cubana de GNU/Linux Nova. El Centro de *software* Libre (CESOL), radicado en la UCI, tiene entre sus objetivos formar personal instruyéndole capacitaciones para realizar procesos de migración del *software* libre y sistemas de código abierto [2].

Nova es la distribución cubana de GNU/Linux desarrollada por estudiantes y profesores de la UCI, para apoyar la migración a las tecnologías de *software* libre; que experimenta Cuba como proceso de informatización de la sociedad. Cuenta con herramientas y bibliotecas de GNU y trabaja bajo el principio de la dependencia de un repositorio propio que tiene como función almacenar y poner a disponibilidad los paquetes de *software* o aplicaciones compatibles con la distribución, actualizaciones de *software*, entre otros servicios [3].

Un buen funcionamiento del repositorio significaría afianzarse ante las empresas cubanas en vías de migración a *software* libre, los usuarios existentes y el aumento de nuevos clientes; así como dar por hecho una mayor calidad en los servicios. En cambio, una mala administración del repositorio de Nova traería consigo un descontento en los usuarios, ya que no dispondrían de los servicios de disponibilidad

de paquetes y sus actualizaciones; lo que podría traer consigo desperfectos en la seguridad y actualización de la distribución. Un repositorio está conformado por paquetes de programas correspondientes a su distribución; estos paquetes deben ser de cabal integridad lo que significa que sean correctamente estructurados, de un formato bien definido, libre de errores, con correcta ejecución funcional y no haber sufrido ataques maliciosos o modificaciones no autorizadas.

En CESOL existe un equipo que tiene como labor la gestión del repositorio de Nova. La forma de detectar errores en los paquetes se ejecuta, por parte de los especialistas, realizando la instalación de cada uno y comprobando su funcionamiento. Esto hace que el procedimiento de verificación de integridad se realice de forma manual, requiera de mucho tiempo de trabajo y en algunos casos sea difícil de ejecutar; lo que conlleva que dicho proceso sea tedioso para los especialistas.

Todo ello trae como consecuencia que en varios casos se cometan errores, no se identifiquen modificaciones y se excluyan paquetes durante la ejecución de pruebas de integridad; por tanto dicho procedimiento es inseguro e incompleto. Debido a la gran cantidad de paquetes del repositorio se hace necesario la verificación de la integridad en cada uno de ellos.

Teniendo en cuenta la situación antes descrita, se identificó el siguiente **problema de investigación**: ¿Cómo comprobar la integridad de los paquetes del repositorio de la distribución cubana de GNU/Linux Nova? Mientras que el **objeto de estudio** del problema planteado lo constituye la integridad de los paquetes en repositorios de distribuciones libres; el **campo de acción** está enfocado en la integridad de los paquetes del repositorio de la distribución cubana GNU/Linux Nova.

Una vez identificado el problema se define como **objetivo general**: Desarrollar una herramienta de pruebas de integridad al repositorio de la distribución cubana GNU/Linux Nova que permita analizar las dependencias de los paquetes, así como detectar la presencia de paquetes averiados.

A partir del objetivo general se determinan los siguientes **objetivos específicos**:

- Elaborar un marco teórico de la investigación sobre el proceso de realización de pruebas de integridad al repositorio de Nova.

- Identificar metodología, las herramientas y las tecnologías de desarrollo de *software* para la solución propuesta.
- Diseñar una herramienta informática para comprobar la integridad de los paquetes del repositorio de la distribución cubana GNU/Linux Nova.
- Implementar la herramienta para la comprobación de la integridad de los paquetes del repositorio de la distribución cubana GNU/Linux Nova.
- Evaluar el correcto funcionamiento del proceso de pruebas de integridad al repositorio de Nova.

A modo de **preguntas científicas**, se plantean las siguientes interrogantes:

- ¿Cómo y cuáles son las disposiciones presentes vinculadas a las herramientas de pruebas de integridad?
- ¿Qué metodología, tecnologías y herramientas se necesitan para agilizar el proceso de realización de pruebas de integridad al repositorio de Nova?
- ¿Qué resultado se alcanza al evaluar la realización de pruebas de integridad al repositorio de Nova?

Para el cumplimiento del objetivo trazado se plantean las siguientes **tareas de investigación**:

- Definición de los conceptos asociados al marco teórico.
- Estudio de la tecnología (lenguajes y herramientas) necesaria para la implementación de la solución.
- Investigación de la existencia de sistemas que realizan pruebas de integridad a repositorios para conocer más detalladamente el funcionamiento de este proceso.
- Realización de los procesos necesarios para la implementación de la propuesta solución.
- Evaluación de los resultados obtenidos durante la investigación.

Para dar soporte a las tareas planteadas anteriormente, se utilizarán los siguientes **métodos científicos** descritos a continuación:

- Análisis-Sintético: es el método que estudia las teorías y documentos necesarios para la extracción de elementos importantes, analizar y determinar las características que deben tenerse en cuenta para la realización de pruebas de integridad sobre el repositorio de la distribución cubana GNU/Linux Nova.
- Histórico-Lógico: es el procedimiento usado para constatar la evolución de las tecnologías en el área de la integridad de los repositorios de distribuciones libres; así como el estudio crítico de antecedentes vinculados a las pruebas de integridad sobre el repositorio, utilizando estos casos como punto de referencia.
- Observación: es la técnica que usa la percepción planificada y consciente dirigida al desarrollo de una herramienta de pruebas de integridad al repositorio de Nova; permite tener un registro de información de funcionalidades y arquitectura que debe tener la solución propuesta.

Para una mejor comprensión del contenido la investigación se estructuró en tres capítulos, conclusiones, recomendaciones, bibliografía utilizada y anexos. Los capítulos se organizan de la siguiente forma:

Capítulo 1. Fundamentación teórica: En el cual se tratan los conceptos y aspectos más significativos abordados en diferentes fuentes bibliográficas, que se relacionan con el proceso de análisis de los paquetes en la Distribución Cubana de GNU/Linux Nova. Se realiza un análisis a diversos sistemas informáticos referentes al tema y se detalla la metodología, los lenguajes y las herramientas utilizadas en la implementación de la propuesta de solución.

Capítulo 2. Descripción de la solución propuesta: Donde se define la propuesta de solución y la arquitectura a utilizar. Se obtienen los requisitos funcionales, no funcionales y se describen mediante el uso de historias de usuarios.

Capítulo 3. Implementación y pruebas: En el que se definen los estándares de codificación para desarrollar una programación homogénea. Se realizan las pruebas internas y de aceptación para identificar posibles fallos. Se aplica la técnica de IADOV para determinar el nivel de satisfacción grupal de los usuarios con la solución propuesta a partir de una encuesta.

Capítulo 1: Fundamentación teórica

1.1 Introducción

Para facilitar la comprensión del alcance de la investigación, en el presente capítulo se aborda un estudio de los fundamentos teóricos, metodológicos y tecnológicos que se ejercen en la actualidad relacionados al proceso de realización de pruebas de integridad al repositorio de Nova. Se realiza un análisis a herramientas de pruebas de integridad a repositorios; así como el estudio de algoritmos que permitan verificar la integridad del repositorio de Nova. También se elabora un estudio de las tecnologías, herramientas y metodología a utilizar durante el desarrollo de la solución propuesta.

1.2 Conceptos asociados

Para lograr una mejor asimilación de la investigación se abordan un conjunto de conceptos necesarios que están estrechamente relacionados con el dominio del problema.

1.2.1 Repositorio

En los sistemas operativos GNU/Linux, un repositorio es una colección de paquetes de programas de una distribución de Linux específica que generalmente contiene archivos binarios, los cuales pueden ser descargados e instalados por los usuarios de la distribución correspondiente. En la actualidad existen diferentes conceptos relacionados con el término; uno de los desarrolladores de Debian, Aaron Isotton, lo define de la siguiente manera: "*Un repositorio es un grupo de paquetes organizados en un árbol de directorios especiales, los cuales contienen también ficheros adicionales que indican los índices y el chequeo de sumas de los paquetes*" [4].

En cuanto a las distribuciones, tomando como ejemplo tanto Ubuntu como Fedora hacen otras menciones en cuanto a repositorio de Linux; definiéndolos como un conjunto de paquetes de *software* disponibles para una determinada distribución almacenados de forma centralizada, estos se encuentran alojados de forma estructurada y organizados por índices de los paquetes que estén disponibles, asegurando su autenticidad e integridad, de manera tal que los usuarios puedan administrarlos usando gestores de

paquetes que los transfieren por vía ftp o http [5;6].

En el caso particular del repositorio de Nova la estructura implementada posee la siguiente descripción:

- **Principal**: El componente principal, contiene los paquetes fundamentales compatibles con todas las versiones de Nova además de las aplicaciones propuestas por la comunidad.
- **Extendido**: Contiene aquellos paquetes que quedaron heredados de otras distribuciones que son compatibles con Nova.
- También cuenta con los archivos *Release, Packages y Sources*, cuyo contenido es información de todos los paquetes del repositorio.

1.2.2 Paquete de *software*

Un paquete de *software* es una colección de archivos de códigos fuente o binario que contienen un conjunto de archivos de instrucciones que especifican funciones a cada uno de ellos. Todos los archivos están comprimidos con un formato específico que depende de la distribución [7]. Los formatos de los paquetes son los siguientes:

Paquetes fuentes: conocidos por usar la extensión ".tgz" aunque también se les conoce como *tarballs*.

- **TGZ**: Es un tipo de archivo para paquetes exclusivos de Unix, comprimido con el compresor GNU Zip. Es un paquete de código fuente, ocupado para contener aplicaciones y su código fuente, para no tener que crear un tipo de paquete específico para cada distribución. La extensión TGZ contiene también a las extensiones como ".tar.gz", ".tar.bz2″. A diferencia de otros paquetes, TGZ, no contiene instrucciones particulares de instalación para cada distribución, por lo que la instalación del contenido deberá ser compilado por el usuario.
- **Ebuild**: Es un *script* construido en el lenguaje de programación Bash, que está estructurado con declaraciones de variables y con las funciones que realizará el paquete. En este tipo de paquetes se puede definir funciones que controlen el proceso de compilación e instalación de paquetes.

Paquetes binarios: Contienen código máquina y no código fuente, por ello cada tipo de procesador requiere de su propia versión de paquete. Cuentan con la información necesaria para reconstruir una aplicación en un sistema nuevo sin necesidad de encontrarse en la misma computadora; los más comunes son:

- **Deb**: Contienen ejecutables, archivos de configuración, páginas de información, derechos de copyright y otras documentaciones; deb es la extensión del formato de paquetes de *software* de Debian y derivadas (por ejemplo: Ubuntu), y el nombre más usado para dichos paquetes. Está estructurado por tres archivos: uno con los archivos a instalar, otro de información de control y el otro de datos como de versión del formato.
- **RPM:** Por sus siglas en inglés *Redhat Package Manager*, este tipo de paquete para Linux fue desarrollado para la distribución de *Red Hat* y sus derivadas (por ejemplos: Fedora, openSUSE, Mandriva), con el fin de crear un sistema fácil de crear e instalar [8].

1.2.3 Integridad

El concepto de integridad está referido a algo puro, completo, intacto, formado por todos sus componentes; también se refiere a una persona culta e intachable, completa [9]. En el ámbito de la informática, la integridad constituye un atributo medidor de un sistema en cuanto a su formato, estructura, capacidad y constitución funcional; es el grado en que es posible controlar el acceso de personas no autorizadas al *software* o a los datos [10]. Por tanto, la integridad puede relacionarse prácticamente con todos los componentes de la informática, resultando un útil indicador en cuanto a la realización de pruebas de calidad.

La integridad de paquetes de un repositorio se define como la característica de capacidad para seguir la pista a todas las relaciones, es vital para el buen funcionamiento de la información almacenada en un repositorio, por ello el proceso de verificación de integridad consiste en el chequeo de datos de los paquetes, evaluar su correspondiente completitud y funcionamiento; examinar si ha sido modificado por error, cortes de comunicación o, en el peor de los casos, porque un atacante se haya infiltrado en el

repositorio.

1.2.4 Pruebas de integridad a paquetes en repositorios

En entornos de código abierto es posible contrastar y compilar el código fuente de un programa, revisándolo para comprobar que no contiene código no deseado. De esta forma se puede garantizar que el *software* no ocasionará ningún daño. Cuando un desarrollador hace público un paquete, suele almacenarlo en un servidor para que sea descargado. Este servidor no siempre está bajo el control del autor del programa o es administrado por la misma persona que ha realizado la herramienta, por tanto, puede ser vulnerable a ataques.

Incluso si el programador aloja el *software* en un servidor administrado por él, es posible que pueda ser comprometido sin su conocimiento y otro atacante tenga la posibilidad de modificar o sustituir el programa que pone a disposición de todos los usuarios. Cuando un atacante se apodera de un servidor web y tiene la posibilidad de modificar un programa que más tarde será descargado por otros, puede suponer un riesgo. Este tipo de ataque se ha dado en muchas ocasiones. Sin las herramientas y precauciones adecuadas, nada garantiza que un programa esté libre de *malware*, ni siquiera el hecho de descargarlo de una página oficial del fabricante.

De cualquier manera, un paquete puede ser modificado: por error, por cortes en la comunicación o en el peor de los casos porque un atacante haya insertado código malicioso; de ahí la necesidad de realizar pruebas de integridad al repositorio de Nova. Comprobar la integridad de los paquetes es la tarea que permite saber a ciencia cierta si estos han sido modificados por personal no autorizado desde su creación, conocer el daño que presentan y si está afectada la funcionalidad de dichos paquetes.

1.3 Análisis a herramientas de pruebas de integridad a un repositorio

Los verificadores de integridad son un subconjunto de los Sistemas de Detección de Intrusos o Intrusiones (SDI) para estaciones de trabajo. Básicamente estos programas se ocupan de comprobar la integridad de los ficheros del sistema donde se ubican. Ya es tradicional que este proceso se realice empleando sumas

de verificación; que son ejecutadas cada cierto intervalo de tiempo y sobre los ficheros seleccionados por el administrador del sistema. Dichos programas suelen verificar los permisos en archivos, directorios y las cuentas de usuarios. De manera general no emiten alarmas, como el resto de los SDI, sino que generan registros con los resultados del trabajo que realizan [11].

A continuación, se realiza un estudio de varias herramientas encargadas de realizar pruebas de integridad en repositorios, a fin de que este sirva de soporte a la materialización del desarrollo de la herramienta.

1.3.1 APT

APT (***Advanced Packaging Tool***): es un sistema de gestión de paquetes que cuenta con un manejo automático de conflictos. Creado por Debian, APT no es un programa en sí, sino que es una biblioteca de funciones C++ de facilidad de uso (para los usuarios que acostumbrados a usar la terminal) que emplea varias líneas de comandos para la atención de paquetes. A continuación, se explica una función relacionadas a la gestión de paquetes a través de APT:

Apt-cache: es una interfaz de líneas de comandos, la misma trabaja directamente sobre la APT permitiendo realizar una serie de operaciones sobre la caché de paquetes. “Apt-cache” no modifica el estado del sistema, pero proporciona operaciones de búsqueda en la información de los paquetes, de las cuales se puede obtener información muy útil; parte de dicha información puede ser las criptografías MD5, SHA256, [12;13].

1.3.2 Reprepro

Reprepro es una herramienta para el manejo local de repositorios de paquetes de Debian. Presenta la capacidad de almacenar ficheros ya sea que se incluyan manualmente o descargados de algún repositorio de *software*. Gestiona los ficheros de configuración y los paquetes de *software* a través de una base de datos, según se compile, por lo que no se necesita un gestor de base de datos convencional. Realiza la gestión de repositorio no limitándose solo a la descarga de los paquetes y creación de la estructura del repositorio, sino que es capaz de realizar la administración y manejo de los índices de los repositorios ubicados de forma local [14].

Entre sus procesos secundarios, cuenta con un sistema para la verificación de los paquetes. Tiene como inconveniente que la base de datos tiende a corromperse cuando el manejo de los repositorios no es el correcto [14].

1.3.3 DSpace

Es un *software* de código abierto que provee herramientas para la administración de repositorios; de amplio uso a nivel mundial, DSpace fue desarrollado en escrito en Java. Usa una base de datos relacional, y soporta el uso de PostgreSQL y Oracle. Cuenta con un conjunto de tareas de curación (*Curation Task*) que básicamente son programas desarrollados en Java para añadir funcionalidades adicionales relacionadas a la gestión de los objetos del repositorio.

Las tareas adoptadas por DSpace son:

- Escaneado antivirus de los ficheros, asegurar la legibilidad de los ficheros.
- Comprobación de la completitud de metadatos, valores límite de los metadatos, adherencia a determinados perfiles de uso de los mismos.
- Conexión con servicios externos a DSpace para mejorar los metadatos.

Los análisis a los objetos del repositorio pueden realizarse a toda la colección de paquetes, o bien sobre un conjunto acotado, una colección especifica; también pueden ser automáticos, semiautomáticos o manuales [15]. Pero esto puede generar una elevada demanda de recursos sobre el servidor que aloja el repositorio durante todo el tiempo de ejecución de los procesos. Para disminuir la demanda se suele seleccionar los recursos a procesar en base a una expresión lógica configurable (ejemplo: según el tipo de dato), también cambiar la forma de ejecución secuencial a fin de obtener un avance uniforme a nivel de recursos en el procesamiento.

- **AuthorityChecker**: Examina la conexión con las autoridades. Es una *Curation Task* que detecta problemas de integridad introducidos por modificaciones realizadas desde las herramientas de gestión, se ejecuta periódicamente, verifica y reporta los casos en los cuales la clave almacenada en un metadato

controlado por un *plugin* de autoridades ha sido eliminada del servicio externo o sea cuando se viola un principio de seguridad e integridad.

- **FileChecker**: Valida los archivos a través de las restricciones que del repositorio. Como resultado de la ejecución de esta tarea se generará un reporte en el cual se incluirán aquellos *ítems* que no cumplan con las restricciones definidas (ejemplo: que contenga un archivo en un formato aceptado) [16].

1.3.4 Koji

Es una herramienta distribuida compuesta por varios nodos que se encargan de la construcción de paquetes; y un nodo principal controlador de todo el proceso, a dicho nodo se conectan los desarrolladores utilizando un cliente para solicitar la construcción de un paquete y obtener información sobre su compilación por lo que de cierta manera se verifica la instalabilidad de los paquetes. La interfaz de esta herramienta es una aplicación web la cual, es de solo lectura [17].

1.3.5 Svnlook

Svnlook es una herramienta desarrollada por del controlador de versiones Subversion[1], para examinar un repositorio. Este programa no intenta en ningún momento cambiar el repositorio ya que es una utilidad de "sólo lectura". Utiliza normalmente por los ganchos (*pre-commit y post-commit)* del repositorio para informar acerca de los cambios que se van a realizar o que se acaban de hacer. Tiene una sintaxis simple, prácticamente cada uno de los comandos puede trabajar sobre una revisión [18].

1.3.6 Aptly

Aptly es considerada una de las herramientas más completas para la gestión de repositorios Debian. Ofrece varias características que facilitan la gestión de los repositorios de paquetes de Debian, destacándose el arreglo de estado de un repositorio. Otras de sus funciones a resaltar son: la verificación de integridad a través del método de firmas digitales, la identificación de paquetes repetidos y las creaciones de repositorios remotos duplicados. Aptly también produce un conjunto fijo de paquetes en el

1 Suvbersion es una herramienta de código abierto utilizada para el control de versiones.

repositorio, haciendo que la instalación y actualización del paquete se vuelva determinista.
Al mismo tiempo, puede realizar cambios controlados y detallados en el contenido del repositorio. Aptly permite hacer la transición del entorno de su paquete a una nueva versión, o revertir a una versión anterior [19].

1.3.7 Ceve

Ceve es un analizador de metadatos generalizado para distribuciones de Ubuntu. Su principal función es leer las especificaciones de los paquetes en repositorios, realizando extracción de información y metadatos de los mismos. De igual forma lleva a cabo algunas manipulaciones y creaciones de metadatos en paquetes de varios formatos. Constituye un analizador muy amplio ya que no se limita solo a especificaciones de los metadatos, por lo que durante su ejecución puede llegar a consumir un número desmedido de recursos computacionales y tiempo [20].

1.3.8 Debsums

Es una herramienta comando cuya función es realizar la suma de comprobación md5 de los paquetes instalados en sistemas Debian y Ubuntu, fue desarrollado en el lenguaje Perl. Debsums verifica la existencia de paquetes corruptos realizando un escaneado de todos los paquetes e informa de los archivos alterados al finalizar la comprobación. La principal desventaja de Debsum resulta en que su uso conlleva a un empleo considerable de tiempo.

1.3.9 Tripwire

Es una aplicación desarrollada en código abierto para sistemas operativos GNU/Linux, que tiene también una versión comercial para Windows. Su funcionamiento consiste en avisar al usuario sobre cualquier cambio en los paquetes del repositorio. El programa crea una base de datos con identificador por cada paquete analizado y puede comparar el momento actual con el de registro en la base de datos, avisando mediante una notificación cualquier alteración, eliminación o inclusión de un archivo en el sistema [22].
Tripwire, permite configurar las reglas o políticas por las que se chequearán los ficheros y directorios. Su notificación consiste en un reporte con el resultado del chequeo realizado a las direcciones que se le

hayan configurado, tan pronto como se detecte una intrusión, lo cual permitirá al administrador tomar acciones que eviten una pérdida de información o un colapso en el sistema. No opera en tiempo real, aunque permite configuraciones para auto ejecutarse cada cierto tiempo [22].

1.3.10 Dose

Es una herramienta que chequea la instalabilidad de los paquetes de acuerdo a la información de los metadatos. Determina si los paquetes se pueden instalar en relación con la distribución correspondiente a su respectivo repositorio. Analiza conjuntos exactos en los datos en los paquetes, los datos de control relevantes y su significado en dependencia del tipo de repositorio. Su algoritmo de resolución de restricciones es completo, es decir, encuentra una solución siempre que exista una, incluso para múltiples dependencias disyuntivas y conflictos de paquetes profundos.

Funciona dividiendo los paquetes en dos planos donde solo los paquetes en primer plano se verifican para determinar su instalación, y las dependencias pueden ser satisfechas por los paquetes los paquetes del segundo plano. Actualmente los tipos de paquetes soportados por Dose son: debian, rpm y eclipse. La herramienta espera sus especificaciones de entrada en un formulario compuesto por: tipo y ruta; donde tipo se refiere a la variedad de paquetes, y ruta de acceso es la dirección de acceso de un archivo de entrada [23]. Dose se establece como una herramienta de pruebas de integridad más completas, por lo que será tomada en cuenta para conformar la propuesta solución.

1.4 Valoración de herramientas de pruebas de integridad a un repositorio

Una vez concluido el estudio de las herramientas para el análisis de repositorios, se puede decir que: APT y Reprepro, aunque implementan mecanismos que están relacionados a la investigación, son consideradas herramientas para la creación de repositorios y no para el análisis de estos. DSpace es una potente herramienta para la administración de repositorios; parte sus *Curation Task* tienen el análisis de paquetes como tareas secundarias. Al igual que Dspace, Aptly constituye una herramienta completa para la gestión de repositorios, puede realizar pruebas vinculadas a la integridad de repositorios destacándose por el aná-

lisis de firmas digitales. Koji se limita al análisis de la instalabilidad y formato de los paquetes. En el caso de Svnlook permite mostrar información sobre las modificaciones realizadas o que se realizarán.

Por último, Ceve, Debsums y Tripwire son compatibles a la solución a implementar; analizan las dependencias que posee cada paquete. Las herramientas antes mencionadas no realizan un estudio completo de los datos para una posterior prueba de integridad; por lo que no son completamente funcionales. Dose, al contrario de las restantes, puede ser tomada en cuenta para la elaboración de la propuesta solución. Por lo antes expuesto se propone desarrollar una nueva aplicación, tomando en cuenta la existencia de la herramienta Dose, contando con las siguientes características a aplicar en la confección de la herramienta solución:

• Conectarse al repositorio GNU/Linux Nova.

• Buscar la información de cada paquete.

• Detectar las inconsistencias que poseen los paquetes.

El estudio de las herramientas de pruebas de integridad permitió determinar los procedimientos capaces de comprobar la integridad de los paquetes en repositorios.

1.5 Métodos prácticos de comprobación de integridad de paquetes en repositorios

A continuación, se muestra una tabla con los métodos de comprobación de la integridad de los archivos y sus respectivas ventajas:

Tabla 1: Ventajas de los métodos de comprobación de integridad de paquetes.

Métodos	**Ventajas**
Funciones *hash* Criptográficas	Garantía de que el archivo no ha cambiado
Firmas Digitales	Garantía de que el archivo no ha cambiado y proviene de una firma digital concreta, aunque nada garantiza que pertenezca a esa persona física determinada.
Certificados Digitales	Garantía de que el archivo no ha cambiado y proviene de una firma digital

	concreta, garantizada su identidad a partir de una tercera persona confiable (Autoridad Certificadora).
Virustotal	Ofrece una idea (no definitiva) sobre si el archivo contiene o no *malware*.

1.5.1 Funciones *hash* criptográficas

Las funciones *hash* son estructuras de datos muy conocidas que se encuentran estrechamente ligadas con la criptografía en general y la integridad de los datos en particular.

Este tipo de funciones convierten un mensaje de cualquier tamaño en un mensaje de una longitud constante. Lo que se obtiene al aplicar una función *hash* criptográfica a un mensaje (flujo de datos, o más usualmente, un archivo) se llama resumen criptográfico, huella digital o *message digest*. Es decir, a partir de un número indeterminado de bits, siempre se obtiene un número constante y diferente que identifica de forma unívoca a ese flujo de datos [24].

Aplicaciones de las funciones *hash*

Con estas premisas, las aplicaciones de las funciones *hash* son claras:

1. Protección de contraseñas: Estas funciones permiten almacenar un resumen criptográfico, en vez del texto claro de las contraseñas. Así, en lugar de comparar las contraseñas en texto claro, se calcula su *hash* con una función y se comparan los resultados. De esta manera, el sistema no tiene por qué almacenar el texto claro en ningún momento para comprobar si alguien conoce una contraseña almacenada.
2. Comprobación de integridad: Si se calcula *hash* de dos flujos de datos y dan un mismo resultado, se puede confirmar que los flujos de datos son idénticos.
3. Garantizar la integridad de un flujo de datos a diferentes niveles:
    - Al descargar u obtener un fichero por cualquier medio, se puede comprobar que se trata del original o que no tiene defectos si el proveedor proporciona un resumen criptográfico para

comparar. Si hubiese cambiado un solo bit del archivo, el resumen sería muy distinto.

- Se puede comprobar si se han producido cambios no controlados en los datos de un sistema de almacenamiento, calculando y comprobando de forma periódica los resúmenes criptográficos de los datos [24].

Algoritmos más utilizados

Los algoritmos más utilizados para calcular el *hash* son:

- **MD5**: Ante la entrada de cualquier flujo de datos, devuelve un bloque de 128 bits. En 2006 se publicó un método capaz de encontrar colisiones en unos minutos y por tanto, aunque muy usado, no se considera totalmente seguro hoy día.
- **SHA256** (y sus sucesores: **SHA512**, por ejemplo): Ante la entrada de cualquier flujo de datos, devuelve un bloque de 256 bits. Al aumentar los bits de salida (hasta 2256 frente a 2128 del MD5), la posibilidad de colisión es menor y por tanto es más seguro. Se utiliza en los principales protocolos de cifrado: SSL, SSH, PGP o IPSec. Los métodos para encontrar colisiones, aunque existen en SHA, no tienen suficiente potencia como para poder proporcionar un ataque práctico, por tanto, se considera relativamente seguro hoy día [24].

1.5.2 Firmas digitales

Gracias a la criptografía simétrica es posible comprobar no sólo que un archivo no ha sido alterado, sino también la autoría por la persona u organización que afirma haberlo creado. Esto se consigue a través de las firmas criptográficas que acompañan a ciertos archivos y, como se ha mencionado, corresponden a la firma del *hash* del archivo. Las firmas suelen ser archivos con extensión SIG o ASC que resultan de firmar criptográficamente con la clave privada del autor, el *hash* de un fichero. Si posteriormente se comprueba, a través de la clave pública, que el fichero firmado concuerda con la firma, es que se está ante un fichero realmente creado por quien dice haberlo hecho, y no modificado desde que se firmó. Esto no garantiza conocer de ningún modo las intenciones o funciones del archivo, sólo su origen [24].

1.5.3 Certificados digitales

Un certificado digital consiste en la asociación entre una entidad física y una firma, realizado por una entidad confiable. Certifica que una firma criptográfica pertenece a una persona, y una entidad lo ha comprobado, es decir, le ha pedido a esa persona sus datos y pruebas de que la firma le pertenece [24].

1.5.4 Virustotal

Siempre existe la posibilidad de un ataque al repositorio, por eso se hace necesario analizar concienzudamente el archivo en busca de virus, troyanos u otro tipo de *malwares*. Es por ello que se hace aconsejable utilizar sistemas de análisis múltiple como *virustotal.com* para asegurar al menos que algunas sistemas antivirus reconocen o no el programa como peligroso. Dicho sistema ofrece la posibilidad de conocer la opinión de múltiples motores antivirus sobre un programa o archivo concreto [24].

1.6 Metodología de desarrollo

La metodología de desarrollo de *software* es el marco usado para estructurar, planear, controlar el proceso de desarrollo de un proyecto que, en dependencia de la plataforma de desarrollo y el lenguaje de programación, utiliza las herramientas que permiten obtener los diferentes artefactos y el producto final. Actualmente no existe una metodología de desarrollo de *software* que sea global, que tenga las características para que se pueda aplicar en cualquier tipo de proyecto. Cada proyecto en correspondencia a su equipo de desarrollo, recursos y requisitos escoge una metodología que se adapte en la mayor medida posible a estas características.

1.6.1 AUP UCI

Para el desarrollo del proyecto se utilizó la metodología AUP UCI (Proceso Ágil Unificado) porque supone una estandarización del proceso de desarrollo de *software* en la UCI. Es la definida para la actividad productiva ya que se adapta al ciclo de vida de los diferentes proyectos de desarrollo de *software*. La investigación se desarrollara durante la fase de Ejecución propuesta por esta metodología y se aplicaran las siguientes disciplinas: Requisitos, Análisis y diseño, Implementación y Pruebas internas. Se utiliza el

escenario 4 que propone utilizar las Historias de usuario como variante de representación de los requisitos.

En dicho escenario es recomendable para el *software* a desarrollar ya que está orientado a proyectos no extensos y una Historia de usuario no contiene información extensa. AUP UCI es una metodología flexible que no requiere de una gran cantidad de desarrolladores. Es concisa en el aspecto de la documentación, permitiendo generar solo la necesaria y no la especificada para cada flujo de trabajo. Está diseñada para trabajar en proyectos pequeños donde la atención se centra en las actividades que realmente son importantes. Permite el uso de herramientas de cualquier tipo, incluyendo aquí las de código abierto, está compuesta por las siguientes fases:

- **Inicio**: Durante el inicio del proyecto se llevan a cabo las actividades relacionadas con la planificación 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 así como 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 [25].

1.7 Lenguajes y herramientas de desarrollo utilizadas

Para el desarrollo de la solución propuesta se realizó un estudio en cuanto al lenguaje de programación y las herramientas a utilizar, teniendo en cuenta las ventajas de cada una y que cumplan con las condiciones del sistema a desarrollar.

1.7.1 Lenguajes de programación

Un lenguaje de programación es un lenguaje diseñado para describir el conjunto de acciones consecutivas que un equipo debe ejecutar. Por lo tanto, un lenguaje de programación es un modo práctico para que los seres humanos puedan dar instrucciones a un equipo. Son herramientas que permiten crear programas, *software* y poner en práctica algoritmos específicos los cuales controlan el comportamiento físico y lógico de una computadora. El lenguaje de programación permite especificar de manera precisa sobre qué datos debe operar un *software* específico, cómo deben ser almacenados o transmitidos dichos datos, y qué acciones debe dicho *software* tomar bajo una variada gama de circunstancias. Todo esto, a través de un lenguaje que intenta estar relativamente próximo al lenguaje humano o natural [26].

Una característica relevante de los lenguajes de programación es precisamente que más de un programador pueda usar un conjunto común de instrucciones que sean comprendidas entre ellos para realizar la construcción de un programa de forma colaborativa; rasgo que, a petición del cliente, estará presente en la elaboración de la propuesta solución [26].

Bash (*Bourne-again shell*): Es un lenguaje interpretado de programación que ayuda al admirador a realizar la mayor parte de las tareas necesarias, tanto en la automatización como en el arranque; su función consiste en interpretar órdenes a través de una consola. Incluye un súper conjunto de instrucciones basadas en la sintaxis del intérprete *Bourne*, con comandos ampliamente difundidos para la serie de sistemas operativos GNU y muchos otros sistemas tipo UNIX, acumulada la experiencia de numerosos intérpretes desarrollados para sistemas de su tipo. Bash es un intérprete de lenguaje de comandos compatible con SH (*Bourne Shell*) que ejecuta comandos desde la entrada estándar o de un archivo [27]. Además, el lenguaje Bash es multiplataforma por lo que puede encontrarse en casi todos los sistemas basados en Unix y MS-DOS.

Se decide utilizar el lenguaje Bash por la familiaridad que tiene el desarrollador con dicho lenguaje. Por otra parte, los comandos de la herramienta Dose, a integrar con la propuesta solución, están implementados en Bash.

Python: Es un lenguaje interpretado de programación, multiplataforma, flexible, su implementación está bajo la licencia de código abierto *Python Software Foundation License*. Python, al ser un lenguaje

interpretado, implica ahorro de tiempo durante el desarrollo de un programa ya que no necesita de compilación. El intérprete puede usarse de modo interactivo, el cual hace fácil el experimentar con las características del lenguaje para probar funciones durante la etapa inicial de desarrollo [28]. Python permite escribir programas muy compactos y legibles, permitiendo ser más pequeños que sus contrapartes en “C” u otros lenguajes por las siguientes razones:

- Lenguaje Interpretado o de *script*: Es aquel que se ejecuta utilizando un programa intermedio llamado intérprete, en lugar de compilar el código a lenguaje de máquina.
- Funciones y bibliotecas: Dispone de un gran cúmulo de funciones incorporadas en el propio lenguaje, permitiendo el tratamiento de cadenas de caracteres, números y archivos. Además, existen librerías que son importadas con facilidad en los programas para tratar temas específicos.
- Detección dinámica del tipo de variable en tiempo de ejecución: Se refiere a que no es necesario declarar el tipo de dato que va a contener una determinada variable, sino que su tipo se determinará en tiempo de ejecución según el tipo de valor al que se asigne, y el tipo de esta variable puede cambiar si se le asigna un valor de otro tipo.
- Sintaxis clara: Posee una sintaxis visual gracias a la notación (con márgenes) de obligado cumplimiento. Además, para distanciar las porciones de código se aconseja tabular, colocando un margen al código que iría dentro de una función o un bucle. Esto beneficia a que todos los programadores adopten unas mismas notaciones y que los programas de cualquier persona tengan un aspecto muy similar.
- Multiplataforma: El intérprete de Python está disponible en multitud de plataformas (Solaris, GNU/Linux, DOS, Windows, OS/2, Mac OS) por lo que si no se utilizan bibliotecas específicas de cada plataforma el programa podrá correr en todos estos sistemas sin grandes cambios.
- Orientado a Objeto: La orientación a objetos es un paradigma de programación en el que los conceptos relevantes se trasladan a clases y objetos del programa. La ejecución del programa consiste en una serie de interacciones entre los objetos [28].

Se decide implementar la propuesta solución en Python a petición del cliente, percibiendo las ventajas de dicho lenguaje con respecto a otros como C#. Además se toma en cuenta de que buena parte de la distribución cubana GNU/Linux Nova esta implementada en Python, por lo que sería conveniente para la compatibilidad de la herramienta con Nova.

UML (Lenguaje Unificado de Modelado): Es lenguaje de modelado de sistemas de *software* más conocido y utilizado en la actualidad. Está diseñado para visualizar, especificar, construir y documentar *software* orientado a objetos. Proporciona "los planos" de un sistema y pueden ser detallados, en función de los elementos que sean relevantes en cada momento.

Es un lenguaje para especificar y no para describir métodos o procesos. Se utiliza para definir un sistema de *software*, para detallar los artefactos en el sistema y para documentar y construir. En otras palabras, es el lenguaje en el que está descrito el modelo. Se puede aplicar en una gran variedad de formas para dar soporte a una metodología de desarrollo de *software* (tal como el Proceso Unificado Racional) [29].

Es el seleccionado para el modelado de la herramienta ya que el desarrollador tiene familiaridad con dicho lenguaje.

1.7.2 Herramientas de desarrollo

Visual Paradigm: es una herramienta que soporta el ciclo de vida completo del desarrollo de *software*: análisis y diseño orientados a objetos, construcción, pruebas y despliegue. Los sistemas de modelado UML ayudan a una más rápida construcción de aplicaciones de calidad y a un menor coste.

Permite dibujar todos los tipos de diagramas de clases, generar código a partir de diagramas, generación de objetos a partir de bases de datos, generación de bases de datos a partir de diagramas de entidad relación y generar documentación, posee licencia gratuita y comercial [30].

Visual Studio Code: es un editor de código fuente o Entorno de Desarrollo Integrado (IDE), en su versión 1.31,0, desarrollado por Microsoft para Windows , Linux y macOS; que incluye soporte para la depuración, control integrado de Git , resaltado de sintaxis, finalización inteligente de código, fragmentos y

refactorización de código. Es gratuito y de código abierto por tanto es personalizable, por lo que los usuarios pueden cambiar el tema del editor, los atajos de teclado y las preferencias.

Visual Studio Code, es compatible con varios lenguajes de programación y un conjunto de características que pueden o no estar disponibles para un idioma dado, como se muestra en la siguiente tabla. Una característica notable es la capacidad de crear extensiones que analizan códigos y herramientas para análisis estático, utilizando el Protocolo de Servidor de Idioma [31].

Git: utilizado en su version 2.1, es una plataforma de desarrollo colaborativa para alojar proyectos de código abierto y gratuito, diseñado para manejar pequeños y grandes programas, con velocidad y eficiencia. El código se almacena de forma pública, aunque también puede hacerse de manera privada, creando una cuenta de pago. Además, Git ofrece herramientas para el trabajo en equipo dentro de un proyecto entre ellas:

- Un sistema de seguimiento de problemas que permiten a los miembros de un equipo detallar un problema con su *software* o una sugerencia que se desee realizar.
- Una herramienta de revisión de código, donde se pueden añadir anotaciones en cualquier punto de un fichero y debatir sobre determinados cambios realizados en un código específico.
- Un visor de ramas donde se pueden comparar los progresos realizados en las distintas ramas de nuestro repositorio.

Gitb es el punto de encuentro para que millones de desarrolladores colaboren en el desarrollo de sus proyectos. Un gran porcentaje de sus repositorios se almacenan en GitHub, y muchos proyectos de código abierto lo utilizan para hospedar su Git, realizar su seguimiento de fallos, hacer revisiones de código y otras operaciones [32].

Jenkins: Es un servidor de integración continua, gratuito, de *software* libre y actualmente uno de los más empleados para esta función. Su principal objetivo son las tareas, donde indicamos qué es lo que hay que hacer, por ejemplo: se puede programar una tarea en la que se compruebe el repositorio de control de versiones cada cierto tiempo, y cuando un desarrollador quiera subir su código al control de versiones, el

software lo compile y se ejecuten las pruebas. Si el resultado no es el esperado o hay algún error, Jenkins notificará al desarrollador, por *email* o cualquier otro medio, para que lo solucione. Si el resultado es correcto, se puede indicar a Jenkins que intente integrar el código y subirlo al repositorio de control de versiones [33; 34].

Jenkins puede trabajar con diversos lenguajes de programación y permite usar un gran número de herramientas de construcción ya sea por soporte propio o mediante *plugins*. Dispone de una interfaz gráfica la cual facilita su uso y configuración mediante formularios *webs*. Además, la herramienta, puede trabajar con el mismo servidor de Jenkins o puede ser desplegada en computadoras esclavas al mismo tiempo que realiza trabajos en paralelo a una mismo ordenador. Algunas de sus características son:

- Comprobación cada cierto periodo de tiempo si se ha realizado algún *commit* en el repositorio de control de versiones (Git), en caso de ser así, compila el código y ejecuta las pruebas.
- Multiplataforma.
- Permite personalizar interfaz.
- Permite variar la manera de notificar errores.
- Integración con bases de datos.
- Permite compilaciones y pruebas distribuidas.
- Comunidad de soporte.
- Integración de correo electrónico, generando reportes o notificaciones [33; 34].

Los servicios de herramienta Jenkins se utilizan por el equipo de desarrolladores de Nova para operaciones llevadas sobre el repositorio y por tanto se aprovecharán como ejecutor de la propuesta solución.

1.8 Conclusiones parciales

El análisis de los conceptos y estado del arte en cuanto a la integridad de los paquetes permitió una visión más amplia de las funcionalidades a desarrollar. Concluido el estudio de varias herramientas se puede constatar que, aunque poseen similitudes en varias de sus funcionalidades, la prioridad es la gestión de

repositorios y entre sus tareas secundarias la revisión de la integridad. Excepto Ceve, Debsums y Tripwire cuyas funcionalidades están orientadas únicamente a la comprobación de la integridad de los paquetes en los repositorios. La solución propuesta será meramente para la comprobación de la integridad de los paquetes de repositorios y para la conformación de una nueva herramienta solo se tomara en cuenta a Dose. Por tanto se decide desarrollar un sistema para la identificación de inconsistencias en los paquetes del repositorio de Nova.

Durante el presente capítulo también se toma para guiar el proceso de ingeniería la metodología AUP UCI en el escenario 4, que provee al desarrollo de la solución con sus respectivas fases, actividades, artefactos y roles; permitiendo estructurar, planificar y controlar el desarrollo de la aplicación. Además, se seleccionaron como lenguajes de programación a Bash, Python y UML para el modelado. Se elige el programa Visual Paradigm para el diseño y modelado, el IDE Visual Studio Code, conjunto al controlador de versiones Git, además de la herramienta Jenkins como instrumento ejecutor de la herramienta a desarrollar.

Capítulo 2: Descripción de la solución propuesta

2.1 Introducción

La planificación y el diseño del *software* son etapas importantes dentro de un proyecto de desarrollo, por lo que resulta clave lograr una buena planificación antes de entrar en el diseño del mismo. En el siguiente capítulo se definen y especifican los requisitos funcionales y no funcionales presentes en el sistema, así como la correcta selección de su arquitectura para organizar y comprender mejor el sistema. El presente capítulo abarca temas referentes a cada una de estas etapas.

2.2 Mapa Conceptual

El Mapa Conceptual brinda una visualización al usuario de los principales conceptos que se manejan en el desarrollo de la propuesta solución. Representa los conceptos fundamentales para el desarrollo de la aplicación y las respectivas relaciones que existen entre ellos. Dichos conceptos esenciales son presentados como clases y las interrelaciones denotan la estructura del funcionamiento, se brinda una mejor comprensión del medio actual para la concepción de un futuro sistema. A continuación, se muestra el modelo conceptual junto a las descripciones de las entidades:

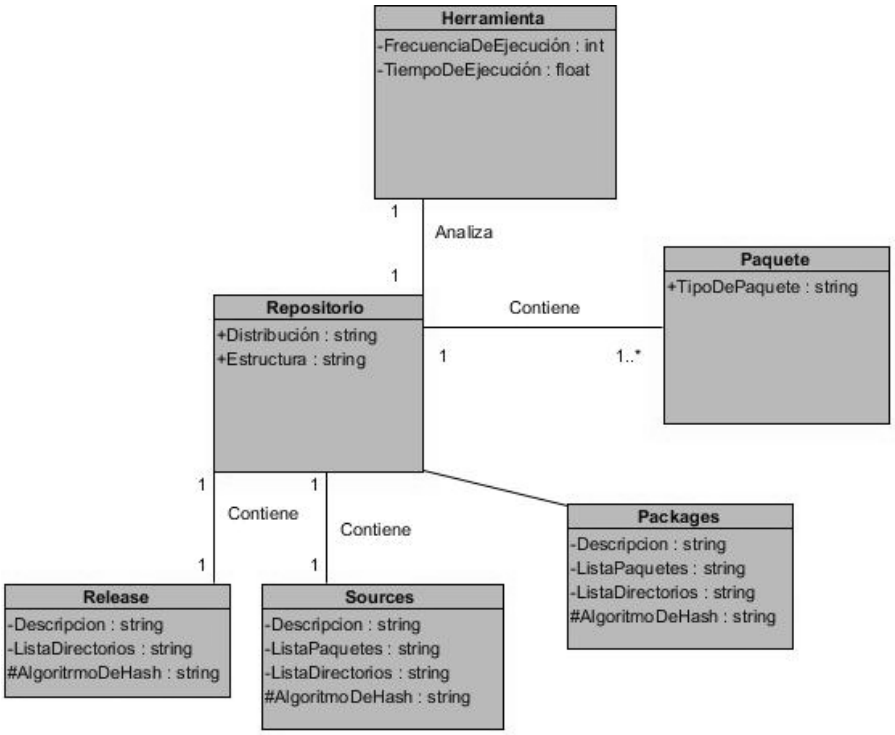


Figura 1: Modelo conceptual.

(Fuente: Elaboración propia)

Herramienta: es la propuesta solución que se lleva a cabo. Consiste en dos scripts que contienen las ordenes necesarias para la comprobación de integridad al repositorio de Nova.

Repositorio: El repositorio de Nova contiene cientos de paquetes de *software* y se encuentra en la red para poner a disposición dichos paquetes.

Paquete: Consiste en una colección de archivos con las instrucciones necesarias para la ejecución de su respectivo *software*. Pueden ser de tipo código fuente o binarios.

Los archivos **Release, Packages y Sources**: guardan información de todos los paquetes del repositorio.

2.3 Requisitos del sistema

Para poder identificar las acciones del sistema y entender su funcionamiento, es fundamental conocer los requisitos funcionales que el sistema debe cumplir. A continuación, se muestran los requisitos funcionales identificados:

2.3.1 Requisitos funcionales

Los requisitos funcionales son declaraciones de los servicios que debe proporcionar el sistema, la manera en que debe reaccionar y comportarse en situaciones particulares [35]. El núcleo del requisito es la descripción del comportamiento requerido, que debe ser clara y concisa. Este comportamiento puede provenir de reglas organizacionales o del negocio, o ser descubiertas por interacción con usuarios, inversores y otros expertos en la organización.

RF-1 Ejecutar herramienta para la comprobación de integridad.

RF-2 Verificar integridad de los índices.

RF-3 Verificar la integridad de los paquetes.

RF-4 Ejecutar herramienta Dose.

RF-5 Verificar instalabilidad de los paquetes.

RF-6 Verificar dependencias de construcción (constructibilidad).

RF-7 Mostrar resultados de herramienta Dose.

RF-8 Generar reportes de herramienta Dose.

2.3.2 Requisitos no funcionales

Los requisitos no funcionales detallan las propiedades o cualidades que el producto debe tener, aumentándole funcionalidad al sistema haciendo al producto atractivo, fácil de usar, rápido y confiable

[36]. Especifican criterios que pueden usarse para juzgar la operación de un sistema en lugar de sus comportamientos específicos, ya que éstos corresponden a los requisitos funcionales. Por tanto, se refieren a todos los requisitos que no describen información a guardar, ni funciones a realizar, sino características de funcionamiento, por eso suelen denominarse atributos de calidad de un sistema.

Según las normas ISO (Organización Internacional de Estandarización), cuyo objetivo principal es guiar el desarrollo de los productos de *software* mediante la especificación de requisitos y evaluación de características de calidad, los requisitos no funcionales pueden ser utilizados en el proceso de mejoras de calidad del producto *software* a desarrollar o como entrada del proceso de evaluación. En otras palabras, los requisitos no funcionales son factores que influyen en la medición de la calidad externa e interna de un *software*; además de que proveen de un conjunto de recomendaciones para realizar la especificación de los requisitos de calidad del producto *software* [37].

Por lo tanto para que las pruebas de integridad tengan suficiente calidad se deben cumplir los requisitos que a continuación se describen:

RnF-1. La herramienta contará de con un método de escaneado para obtener los datos referentes a la integridad de los paquetes y analizar dichos datos, por lo que será eficiente durante su funcionamiento. El sistema tendrá la capacidad de procesar los datos con un rendimiento esperados mediante el uso de cantidades y tipos adecuados de recursos en un contexto de uso específico.

RnF-2. Los datos pueden ser obtenidos por usuarios en un contexto de uso específico. Las funcionalidades de la herramienta estarán disponibles en todo momento pero solo podrán ser accedidas por el administrador del repositorio. Será accesible por los usuarios autorizados en todo momento.

RnF-3. La solución propuesta tendrá la capacidad de desempeñar la función requerida, en condiciones establecidas durante un período de tiempo determinado. Es decir, la herramienta es confiable, cumple su objetivo y en el momento que operen sus funcionalidades. Esta característica también incluye el continuo funcionamiento del programa inclusive en la parecencia de errores.

RnF-4. La herramienta tendrá como base la facilidad de uso por las personas autorizadas. Este requisito se describe como la capacidad del producto para ser entendido, aprendido, usado y resultar atractivo

para el usuario, cuando se usa bajo determinadas condiciones. Esta característica se fragmenta en las siguientes cualidades:

- Capacidad para reconocer su adecuación: Es la capacidad del producto que permite al usuario entender si el *software* es adecuado para sus necesidades.
- Capacidad de aprendizaje: Es la capacidad del producto que permite al usuario aprender su aplicación.
- Capacidad para ser usado: Se presenta como la capacidad del producto que permite al usuario operarlo y controlarlo con facilidad.
- Protección contra errores de usuario: Se define como la capacidad del sistema para proteger a los usuarios de hacer errores.

La propuesta solución será fácil de usar, por lo que no se necesita ser un usuario avanzado para darle funcionamiento.

RnF-5. La funcionalidad representa la capacidad del *software* para proporcionar funciones que satisfacen las necesidades declaradas e implícitas, cuando el producto se usa en las condiciones especificadas. Esta característica se fracciona en los siguientes rasgos:

- Completitud funcional: Grado en el cual el conjunto de funcionalidades cubre todas las tareas y los objetivos del usuario especificados.
- Corrección funcional: Capacidad del producto o sistema para proveer resultados correctos con el nivel de precisión requerido.
- Pertinencia funcional: Capacidad del producto *software* para proporcionar un conjunto apropiado de funciones para tareas y objetivos de usuario especificados.

La herramienta a desarrollar cumplirá con todos los grados funcionales requeridos por el cliente.

RnF-6. La propuesta solución, en su funcionamiento normal, tendrá la capacidad de ser provisto de mantenimiento y soporte ante posibles tipos de fallos. Esta característica representa la capacidad del producto *software* para ser modificado efectiva y eficientemente, debido a necesidades evolutivas, correctivas o perfectivas. Esta característica se subdivide a su vez en las siguientes subcaracterísticas:

- Reusabilidad: La herramienta tiene la capacidad de un activo que permite que sea utilizado en más de un sistema *software* o en la construcción de otros activos.
- Analizabilidad: El sistema presenta facilidad para evaluar el impacto de un determinado cambio sobre el resto del *software*, diagnosticar las deficiencias o causas de fallos en el *software*, o identificar las partes a modificar.
- Capacidad para ser modificado: La solución propuesta cuenta con la capacidad de ser modificado de forma efectiva y eficiente sin introducir defectos o degradar el desempeño.
- Capacidad para ser probado: El producto tiene la facilidad para establecer criterios de prueba en sí mismo como sistema o componente, y a su vez llevar a cabo las pruebas para determinar si se cumplen dichos criterios [37].

Debido a que los *software* constantemente tienden a evolucionar, como característica propia de las tecnologías, la herramienta tendrá la capacidad de contar con un soporte que le brinde mantenimiento. Dicha tarea puede ejecutarse por cualquier medio del equipo de desarrollo de Nova o inclusive por personal ajeno al centro ya que la herramienta será de código abierto y por tanto se podrá modificar.

2.3.3 Validación de los requisitos

La validación de requisitos tiene gran importancia, ya que los errores en el documento pueden conducir a grandes costos, como repetir el trabajo cuando son descubiertos durante el desarrollo o después de que el sistema esté en uso. Garantiza que todos los servicios del sistema han sido enunciados sin ambigüedades; que se detectaron y corrigieron las inconsistencias, las omisiones y los errores, y que los productos del trabajo se presentan conforme a los estándares establecidos para el proceso, el proyecto y el producto [35]. A continuación, se describen las técnicas de validación de requisitos empleadas:

Revisiones de requerimientos: Una revisión de requerimientos es un proceso manual, informal o formal que involucra tanto a clientes como desarrollador. Los requerimientos son analizados sistemáticamente por un equipo de revisores. Ellos verifican el documento de requerimientos en cuanto a anomalías y omisiones. Los conflictos, contradicciones, errores y omisiones en los requerimientos deben ser señalados por los revisores y registrarse formalmente [38].

Construcción de prototipos: La construcción de prototipos permite mostrar un modelo ejecutable del sistema a los usuarios finales y a los clientes. Estos pueden experimentar con este modelo para ver si cumple sus necesidades reales [38].

2.4 Historias de Usuario

La Historia de Usuario es una técnica utilizada para caracterizar y detallar los requisitos de *software*. Constituyen parte central del escenario 4 de la metodología AUP Variación para la UCI, pues definen la construcción del proyecto de *software*, guían la construcción de las pruebas de aceptación y son utilizadas para estimar tiempos de desarrollo. Deben estar detalladas a través de la comunicación con el cliente, pues constituyen una base para las pruebas funcionales [39]. A continuación, se describen las historias de usuario para el desarrollo de la herramienta:

Tabla 2: Historia de usuario #1: Ejecutar herramienta para la comprobación de integridad.

Historia de Usuario
Número: RF-1
Nombre de historia: Ejecutar herramienta para la comprobación de integridad.
Usuario: Administrador
Prioridad: Alta
Riesgo en desarrollo: Fallos en la etapa de ejecución, por consiguiente, será imposible que se realice el resto de los procesos.
Tiempo estimado: 3 semanas.
Programador responsable: Pedro Javier Hernández Melgarejo.
Descripción: Inicia la ejecución de la herramienta. La funcionalidad permite que se realicen los procedimientos de verificación de integridad en el repositorio.

Tabla 3: Historia de usuario #2: Verificar integridad de los paquetes.

Historia de Usuario
Número: RF-2
Nombre de historia: Verificar integridad de los índices.
Usuario: Administrador
Prioridad: Alta
Riesgo en desarrollo: Fallos en la etapa de verificación en el índice, por tanto, se obtendrán resultados incorrectos. Imposibilidad en el reconocimiento del archivo Release.
Tiempo estimado: 4 semanas.
Programador responsable: Pedro Javier Hernández Melgarejo.
Descripción: Realiza verificaciones de integridad a los índices teniendo como entrada las rutas en la publicación del archivo Release. La funcionalidad permite al usuario, verificar el contenido del archivo Release y los índices o relaciones del repositorio.
Observaciones: Esta prueba se realiza para verificar que coinciden las informaciones de ubicación de los paquetes. De esta manera permite chequear los archivos y directorios de cada paquete.

Tabla 4: Historia de usuario #3: Verificar la integridad de los paquetes.

Historia de Usuario
Número: RF-3
Nombre de historia: Verificar la integridad de los paquetes.
Usuario: Administrador
Prioridad: Alta
Riesgo en desarrollo: Errores en verificación de artefactos en el índice.
Tiempo estimado: 4 semanas.
Programador responsable: Pedro Javier Hernández Melgarejo.
Descripción: Realiza verificaciones de integridad a los paquetes teniendo como entrada los identificadores de cada paquete y el archivo Release. Esta función permite al usuario chequear los archivos y directorios, a través de la comprobación de las sumas de los algoritmos criptográficos MD5, SHA256, SHA512 para cálculo del *hash*. En caso de que no haya ningún paquete alterado se indica que en el repositorio no hay paquetes con inconsistencias.
Observaciones: Esta prueba determina si un paquete del repositorio ha sufrido modificaciones.

Tabla 5: Historia de usuario #4: Mostrar resultados de herramienta.

Historia de Usuario
Número: RF-4
Nombre de historia: Mostrar resultados de herramienta.
Usuario: Administrador
Prioridad: Alta
Riesgo en desarrollo: Fallos al mostrar los paquetes deteriorados.

Tiempo estimado: 1 semana.
Programador responsable: Pedro Javier Hernández Melgarejo.
Descripción: Se muestran los paquetes deteriorados junto a sus tipos de errores. Además, se revelan datos de cantidad de paquetes del repositorio. Se pondrá en marcha un código de la herramienta Dose sobre el repositorio y se obtiene los resultados correspondientes a las dependencias de construcción en cada paquetes. El usuario podrá chequear las dependencias de los paquetes y garantizar la ejecución recursiva del chequeo de los mismos.

Tabla 6: Historia de usuario #5: Ejecutar la herramienta Dose.

Historia de Usuario
Número: RF-5
Nombre de historia: Ejecutar la herramienta Dose.
Usuario: Administrador
Prioridad: Alta
Riesgo en desarrollo: Fallos en la etapa de ejecución, por consiguiente, será imposible que se realice el resto de los procesos vinculados a la constructibilidad e instalabilidad.
Tiempo estimado: 4 semanas.
Programador responsable: Pedro Javier Hernández Melgarejo.
Descripción: Inicia la ejecución de la herramienta. La funcionalidad permite que se realicen los procedimientos de verificación de constructibilidad e instalabilidad en el repositorio.

Tabla 7: Historia de usuario #6: Verificar la instalabilidad de los paquetes.

Historia de Usuario
Número: RF-6

Nombre de historia: Verificar la instalabilidad de los paquetes.
Usuario: Administrador
Prioridad: Alta
Riesgo en desarrollo: Errores en el reconocimiento de los paquetes.
Tiempo estimado: 3 semanas.
Programador responsable: Pedro Javier Hernández Melgarejo.
Descripción: Realiza verificaciones de instalabilidad de los paquetes tomando como entrada las direcciones de publicación para cada paquete. La herramienta ejecutará un comando de la herramienta Dose sobre el repositorio y se obtendrá el resultado respecto a la instalabilidad de cada paquete.

Tabla 8: Historia de usuario #7: Verificar las dependencias de construcción de los paquetes.

Historia de Usuario
Número: RF-7
Nombre de historia: Verificar las dependencias de construcción de los paquetes.
Usuario: Administrador
Prioridad: Alta
Riesgo en desarrollo: Errores en el reconocimiento de los paquetes.
Tiempo estimado: 4 semanas.
Programador responsable: Pedro Javier Hernández Melgarejo.
Descripción: Realiza la verificación de las dependencias de construcción de los paquetes del repositorio tomando como referencia la ubicación de su publicación. Se pondrá en marcha un código de la herramienta Dose sobre el repositorio y se obtiene los resultados correspondientes a las dependencias de construcción en cada paquetes. El usuario podrá chequear las dependencias de los paquetes y garantizar la ejecución recursiva del chequeo de los mismos.

Tabla 9: Historia de usuario #8: Mostrar resultados de herramienta Dose.

Historia de Usuario
Número: RF-8
Nombre de historia: Mostrar resultados de herramienta Dose.
Usuario: Administrador
Prioridad: Alta
Riesgo en desarrollo: Fallos al mostrar los paquetes deteriorados.
Tiempo estimado: 1 semana.
Programador responsable: Pedro Javier Hernández Melgarejo.
Descripción: Se muestran los paquetes deteriorados junto a sus tipos de errores. Además, se revelan datos de cantidad de paquetes del repositorio. Se pondrá en marcha un código de la herramienta Dose sobre el repositorio y se obtiene los resultados correspondientes a las dependencias de construcción en cada paquetes. El usuario podrá chequear las dependencias de los paquetes y garantizar la ejecución recursiva del chequeo de los mismos.

Tabla 10: Historia de usuario #9: Mostrar resultados de herramienta Dose.

Historia de Usuario
Número: RF-9
Nombre de historia: Generar reportes de la herramienta Dose.
Usuario: Administrador
Prioridad: Alta
Riesgo en desarrollo: Fallos en el tipo de formato de los reportes a generar.
Tiempo estimado: 2 semanas.

Programador responsable: Pedro Javier Hernández Melgarejo.
Descripción: Se generan varios reportes clasificados según el tipo de paquetes.

2.5 Definición de la arquitectura

Los estilos arquitectónicos definen la estructura de un *software* los cuales a su vez se componen de subsistemas con sus responsabilidades; también tienen una serie de directivas para organizar los componentes del mismo con el objetivo de facilitar la tarea del diseño de tal sistema. La arquitectura definida para la solución propuesta es Flujo de datos, la cual se aplica cuando los datos de entrada se habrán de transformar en datos de salida mediante una serie de componentes computacionales o manipulaciones [40]. La arquitectura propuesta permite operar en entornos distribuidos con un nivel de abstracción superior, además de lograr una interfaz de usuario más flexible y permitir que la aplicación sea más simple y escalable.

2.6 Propuesta solución

Mediante la verificación de integridad de los repositorios se pueden detectar errores ocurridos cuando se publicaron los repositorios y asegura que cada paquete contenido en los índices esté presente en la ruta correcta y con la suma de verificación adecuada. Para lograr esto es necesario verificar la integridad de los índices (comprobando que estén firmados adecuadamente) y los paquetes en el repositorio. Para poder proteger la integridad de los paquetes y la confianza hacia nuestros desarrolladores, se decide desarrollar un *script* como mecanismo que guarde las instrucciones, que posteriormente empleará un interprete de ordenes mas avanzado.

En el mecanismo propuesto se utilizan los algoritmos MD5Sums, SHA256 y SHA512 que proporciona la herramienta Dose, comprobando el código *hash* único para cada fichero. Incluso para un mismo fichero, con solo adicionar un simple espacio este código cambia completamente; facilitándole al sistema comprobar la integridad de sus bibliotecas comparando de cada una de ellas los códigos *hash* generados

en tiempo de ejecución y compilación. Para la ejecución del *script,* se propone el uso del servidor de automatización Jenkins como el interprete de órdenes avanzado.

Jenkins, como instrumento de ayuda en la automatización de parte del proceso de desarrollo de programas, fue publicado bajo licencia de *software* libre y tiene la capacidad de ejecutar proyectos basados en Apache[2], *script*s y programas bash. La herramienta de pruebas de integridad, será capaz de comprobar la integridad de los paquetes del repositorio de GNU/Linux Nova y de esta manera detectar las posibles inconsistencias en dicho escenario. Como resultado final se le brinda, al usuario, información acerca del estado de los paquetes del repositorio.

2.7 Conclusiones parciales

Durante el capítulo se llevó a cabo el proceso de análisis y diseño; en el cual se realizó la elaboración de el Mapa Conceptual que permite representar las entidades relacionadas dando mejor entendimiento de la solución propuesta. Se llevo a cabo la identificación de los requisitos funcionales y no funcionales. Se modeló, como parte de la metodología, las Historias de Usuario correspondientes a la solución. La arquitectura y patrón arquitectónico seleccionados están a tono en relación a las características de la herramienta a desarrollar. Hasta el momento, la investigación realizada ha permitido que el *software* esté prácticamente listo para su implementación.

2 Apache es un servidor web HTTP de código abierto utilizado para enviar paginas web estáticas y dinámicas en ala red informática mundial.

Capítulo 3: Implementación y Pruebas

3.1 Introducción

En la etapa de implementación de un *software*, se transforman las clases y objetos en ficheros fuente, binarios y ejecutables. El resultado final es un sistema que durante la etapa de pruebas, se evalúa su calidad y desempeño como producto de *software*. Probar es imprescindible para verificar el adecuado funcionamiento de un *software*. La prueba es un proceso de ejecución de un programa con la intención de comprobar que el producto satisface los requerimientos y se comporta de la forma deseada. En esta etapa se detectan y corrigen errores para la posterior aceptación del producto. El presente capítulo aborda los temas relacionados con la implementación y las pruebas realizadas a la herramienta de pruebas de integridad al repositorio de la distribución cubana GNU/Linux Nova.

3.2 Estándares de codificación

Conocidos como codificación por convención o simplemente estándares de codificación, son un paradigma de programación que busca reducir el número de decisiones que el desarrollador tiene que tomar al momento de escribir su código. En otras palabras tienen como principal función hacer que la actividad de codificar se vuelva más sencilla e incluso se pueda volver intuitivo; pues la forma en la que se define cada parte del código es utilizada por todos los programadores de un proyecto, así el estándar se comparte y se aprende más rápido, facilitando la lectura y escritura de código. Son utilizados tanto en proyectos de *software* libre como en privados [41; 42].

Para el presente proyecto se definen estándares de codificación porque un estilo de programación uniforme que permite una comprensión del código en menos tiempo; y en consecuencia que el programa le puedan realizar operaciones de mantenimiento. Es por ello que se definen los siguientes estándares de codificación:

Número de declaraciones por línea

Se puede declarar cada variable en una línea distinta, de esta manera cada variable se puede comentar por separado.

Inicialización

Se puede inicializar cada variable en su declaración a menos que su valor dependa de algún cálculo.

Sentencias simples

Cada línea debe contener una sola sentencia.

Líneas en blanco

Se pueden usar dos líneas en blanco entre diferentes secciones de un fichero de código fuente. También se puede usar una linea en blanco entre:

- los métodos.
- las variables locales de un método y la primera sentencia.
- diferentes secciones lógicas dentro de un fichero.

Caracteres de espacio

Se deben usar en las siguientes circunstancias:

- Después de una coma en la lista de parámetros de un método.
- Entre operadores binarios: +, -, =, <, >.
- En las sentencias *for, while* entre sus parámetros [41; 42].

3.3 Pruebas de *software*

Todo sistema de cómputo al ser desarrollado debe pasar por una serie de pruebas de *software* para erradicar las no conformidades que se encuentren en el producto y validar su funcionamiento. El instrumento adecuado para determinar el estado de la calidad de un producto de *software* es el proceso de pruebas. Los exámenes de *software* son un concepto que a menudo, es conocido como verificación y validación. 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* [35]. Este no es un proceso que se realiza una vez desarrollado el *software*, sino que debe efectuarse en cada una de las etapas de desarrollo. La creciente inclusión del *software* como un elemento más de muchos sistemas y la importancia de los costos

asociados a un fallo del mismo han motivado la creación de pruebas más minuciosas y bien planificadas.

3.3.1 Pruebas internas

Las pruebas internas permiten verificar el resultado de la implementación. Se enmarcan en la lógica de procesamiento interno y en las estructuras de datos como: el código fuente y los archivos de datos. Como parte del proceso de pruebas internas que se llevó a cabo se encuentra la técnica caja blanca. Dicho procedimiento es una filosofía de diseño de casos de prueba que usa la estructura de control descrita como parte del diseño a nivel de componentes para derivar casos de prueba.

Al usar los métodos de prueba de caja blanca, puede derivar casos de prueba que: 1) garanticen que todas las rutas independientes dentro de un módulo se revisaron al menos una vez, 2) revisen todas las decisiones lógicas en sus lados verdadero y falso, 3) ejecuten todos los bucles en sus fronteras y dentro de sus fronteras operativas y 4) revisen estructuras de datos internas para garantizar su validez [35].

Prueba del camino básico

El método del camino básico es una técnica de prueba de caja blanca, que permite obtener una medida de la complejidad de un diseño procedimental, y utilizar esta medida como guía para la definición de una serie de caminos básicos de ejecución, diseñando casos de prueba que garanticen que cada camino se ejecuta al menos una vez. La idea es derivar casos de prueba a partir de un conjunto dado de caminos independientes por los cuales puede circular el flujo de control [35]. Para obtener dicho conjunto de caminos independientes se construye el Grafo de Flujo asociado y se calcula su complejidad ciclomática. Los pasos que se siguen para aplicar esta técnica son:

1. A partir del diseño o del código fuente, se dibuja el grafo de flujo asociado.
2. Se calcula la complejidad ciclomática del grafo.
3. Se determina un conjunto básico de caminos independientes.
4. Se preparan los casos de prueba que obliguen a la ejecución de cada camino del conjunto básico.

Los componentes son:

Nodo: Cada círculo representado se denomina nodo del Grafo de Flujo, el cual representa una o más

secuencias procedimentales. Un solo nodo puede corresponder a una secuencia de procesos o a una sentencia de decisión. Puede ser también que hallan nodos que no se asocien, se utilizan principalmente al inicio y final del grafo.

Aristas: Las flechas del grafo se denominan aristas y representan el flujo de control, son análogas a las representadas en un diagrama de flujo. Una arista debe terminar en un nodo, incluso aunque el nodo no represente ninguna sentencia procedimental.

Regiones: Las regiones son las áreas delimitadas o polígonos,por las aristas y nodos. También se incluye el área exterior del grafo, contando como una región más. Las regiones se enumeran. La cantidad de regiones es equivalente a la cantidad de caminos independientes del conjunto básico de un programa.

Complejidad ciclomática: es una medición útil para aquellos módulos proclives al error así como para predecir la complejidad lógica de un programa. La complejidad ciclomática V(G) se calcula en una de tres formas:

1. El número de regiones del gráfico de flujo corresponde a la complejidad ciclomática.
2. V(G) = A - N + 2 donde A es el número de aristas del gráfico de flujo y N el número de nodos del gráfico de flujo.
3. V(G) = P + 1 donde P es el número de nodos predicado (nodos con más de una arista de salida) contenidos en el gráfico de flujo G [35].

En la siguiente figura se muestra los comandos de un método perteneciente a la propuesta solución:

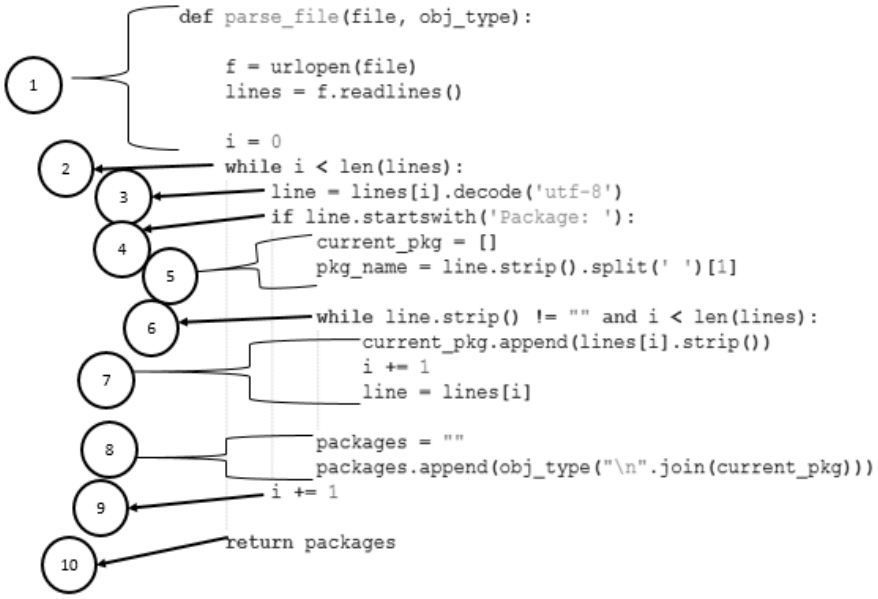


Figura 2: Comandos del método parse_file .

(Fuente: Elaboración propia)

A continuación, se muestran los resultados arrojados por las pruebas de camino básico al método parse_file:

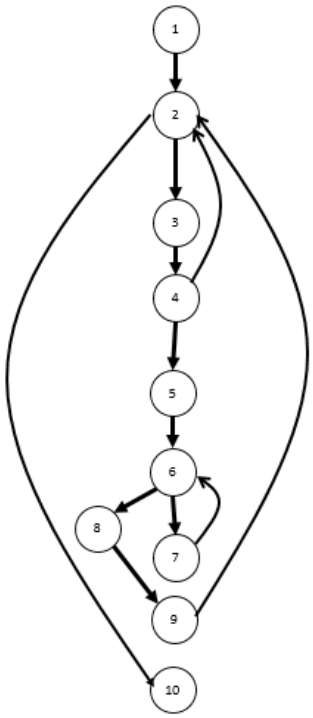


Figura 3: Diagrama de flujo del método parse_file.

(Fuente: Elaboración propia)

A continuación, el calculo de la complejidad ciclomática:

V(G) = 4 , el grafo tiene 4 regiones.

V(G) = A – N + 2 = 12 – 10 + 2 = 4 regiones.

V(G) = P + 1 = 3 + 1 = 4 regiones.

A continuación, la representación de los caminos básicos:

Camino básico #1: 1; 2; 10.

Camino básico #2: 1; 2; 3; 4; 2; 10.

Camino básico #3: 1; 2; 3; 4; 5; 6; 8; 9; 2; 10.

Camino básico #4: 1; 2; 3; 4; 5; 6; 7; 6; 8; 9; 2; 10.

Casos de prueba

Después de haber extraído los caminos básicos, se procede a ejecutar los casos de prueba. Los casos de prueba se derivan para asegurar que todos los enunciados del programa se ejecutaron al menos una vez durante las pruebas y que todas las condiciones lógicas se revisaron. En los casos de prueba se tiene un conjunto de entradas, condiciones de ejecución y resultados esperados desarrollados para un objetivo particular como, por ejemplo; ejercitar el camino concreto de un programa o verificar el cumplimiento de un requisito determinado [35].

Tabla 11: Caso de prueba #1.

Caso de Prueba 1	
Descripción de la prueba	Se desea mostrar el contenido del archivo, teniendo como base un archivo nulo.
Condición de ejecución	El repositorio debe estar conectado y en funcionamiento. El *script* debe estar en ejecución.
Entrada	Se pasa como parámetro un archivo (file), que en este caso será nulo, y el tipo de archivo (obj_type).
Resultado esperado	No se devuelve ningún resultado ya que el file que se introdujo por parámetro es nulo.
Evaluación de prueba	El camino se ha recorrido satisfactoriamente.

Tabla 12: Caso de prueba #2.

Caso de Prueba 2

Descripción de la prueba	Se quiere mostrar el contenido de los archivos, teniendo como base un archivo altamente codificado.
Condición de ejecución	El repositorio debe estar conectado y en funcionamiento. El *script* debe estar en ejecución.
Entrada	Se pasa como parámetro un achivo (file) y el tipo de archivo (obj_type). El file ha sido codificado por un *software* privativo.
Resultado esperado	En este caso el file no podrá ser leído o decodificado por la comando decode('utf-8'); ya que la librería de decodificación no soporta la codificación que posee el file. Por tanto no se devuelve ningún resultado.
Evaluación de prueba	El camino se ha recorrido satisfactoriamente.

Tabla 13: Caso de prueba #3.

Caso de Prueba 3	
Descripción de la prueba	Se quiere mostrar el contenido de los archivos, teniendo como base un archivo que contiene un formato protegido, de solo lectura.
Condición de ejecución	El repositorio debe estar conectado y en funcionamiento. El *script* debe estar en ejecución.
Entrada	Se pasa como parámetro un archivo (file) y el tipo de archivo (obj_type). El file solo tiene permiso de lectura, por lo que no se podrá copiar su contenido.
Resultado esperado	Al file no se le podrán copiar los contenidos. Por tanto se devolverá un paquete nulo.
Evaluación de prueba	El camino se ha recorrido satisfactoriamente.

Tabla 14: Caso de prueba #4.

Caso de Prueba 4	
Descripción de la prueba	Se quiere mostrar el contenido de los archivos, teniendo como base un archivo común.
Condición de ejecución	El repositorio debe estar conectado y en funcionamiento. El *script* debe estar en ejecución.
Entrada	Se pasa como parámetro un archivo (file) y el tipo de archivo (obj_type). En este caso el file no contiene ningún tipo de alteración.
Resultado esperado	Se devolverá un paquete con sus respectivos datos.
Evaluación de prueba	El camino se ha recorrido satisfactoriamente.

Para una primera iteración de la prueba fueron detectadas tres no conformidades explicadas a continuación:

- Una vez ejecutada la herramienta, la ruta o directorio del repositorio no se reconoció.
- Durante la ejecución del proceso de prueba, al no existir un paquete, la herramienta detuvo su funcionamiento.
- Una vez concluida la ejecución de la herramienta, los resultados mostrados no eran comprensibles ya que no tenían un orden coherente.

En la realización de una segunda iteración fueron detectados dos no conformidades expuestas a continuación:

- Si se modifica o elimina mas de un paquete del repositorio, en la ejecución de la herramienta solo se reconoce uno de los paquetes.
- Al finalizar la ejecución de la prueba no se detallan los errores encontrados.

Durante una tercera iteración de la prueba se detecto la no conformidad que se describe a continuación:

- En la ejecución del proceso de prueba si se encontraba un paquete modificado la herramienta

detenía su funcionamiento.

En una cuarta iteración no fueron encontradas no conformidades, por lo cual no fue necesaria la ejecución de una cuarta iteración.

La siguiente gráfica describe la cantidad no conformidades detectados por cada iteración en búsqueda de errores:

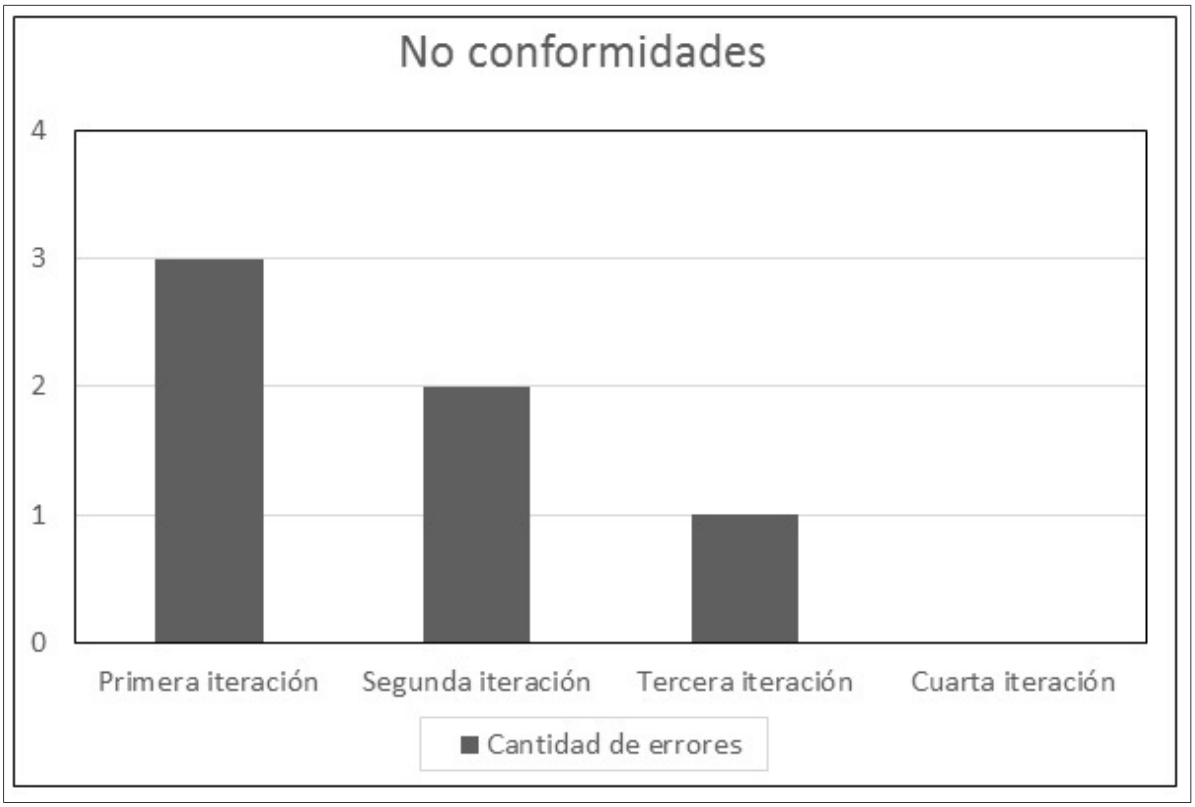


Figura 4: Gráfica con resultados de las iteraciones

3.4 Técnica del cuestionario de satisfacción grupal (IADOV)

La técnica de criterio de usuarios se usa como vía para valorar resultados en aquellos casos en que los evaluadores son usuarios de la solución que se propone, es decir que además de tener dominio del problema en estudio, están contextualizados [43]. Es por ello que para medir la satisfacción del cliente se puso en práctica la técnica IADOV a través de una encuesta a un grupo de desarrolladores de CESOL (Ver Anexo 2). Esta vía calcula el Índice de Satisfacción Grupal (ISG) que se implementa mediante un cuestionario en el cual se le incluyen tres preguntas cerradas de un total de cinco cuestiones; a partir de las respuestas se realiza el Cuadro Lógico de IADOV.

Tabla 15: Cuadro Lógico de IADOV.

4. Luego de haber visto la automatización del proceso de verificación de la integridad en los paquetes del repositorio de la distribución cubana GNU/Linux Nova, represente en que nivel le gusta la solución desarrollada.	2. ¿Considera usted correcta la forma en que se realiza el proceso de verificación de la integridad en los paquetes del repositorio de la distribución cubana GNU/Linux Nova?								
	No			No			Sí		
	3. ¿Considera factible contar con una aplicación informática que permita detectar los paquetes averiados ?								
	Sí	No sé	No	Sí	No sé	No	Sí	No sé	No
Me gusta mucho	1	2	6	2	2	6	6	6	6
Me gusta mas de lo que me disgusta	2	2	3	2	3	3	6	3	6
Me da lo mismo	3	3	3	3	3	3	3	3	3
Me disgusta mas de lo que me gusta	6	3	6	3	4	4	3	3	4
No me gusta nada	6	6	6	6	4	4	6	6	5
No se decir	2	3	6	3	3	3	6	6	4

El número de la interrelación de las tres preguntas indica la posición en la escala de satisfacción siguiente: clara satisfacción (A), más satisfecho que insatisfecho (B), no definida (C), más insatisfecho que satisfecho (D), clara insatisfacción (E) y contradictoria [42].

Para obtener el índice de satisfacción grupal ISG se trabaja con los diferentes niveles de satisfacción que se expresan en la escala numérica que oscila entre +1 y – 1. El número resultante de la interrelación de las tres preguntas indica la posición de cada encuestado en la siguiente escala de satisfacción:

Tabla 16: Escala de satisfacción.

+1	Máxima satisfacción
+0,5	Más satisfecho que insatisfecho
0	No definido y contradictorio
-0,5	Más insatisfecho que satisfecho
-1	Máxima insatisfacción

Como se puede observar en la tabla anterior el índice general arroja valores entre + 1 y - 1. Los valores que se encuentran comprendidos entre -1 y – 0.5 indican insatisfacción; los comprendidos entre – 0.49 y +0.49 evidencian contradicción y los que están entre 0.5 y 1 indican que existe satisfacción [42].

A partir de la cantidad de respuestas por categoría es posible calcular el ISG siguiendo la siguiente fórmula:

$$ISG = \frac{A\,(+1) + B\,(+0.5) + C\,(+0) + D\,(-0.5) + E(-1)}{N}$$

Donde N es la cantidad total de respuestas.

En la siguiente tabla se representan los resultados obtenidos de la aplicación de la encuesta a los especialistas del centro de CESOL, teniendo en cuenta que fue aplicada a 7 especialistas:

Tabla 17: Resultados de la escala de satisfacción.

Categorías grupales de satisfacción	Cantidad de respuestas. N = 7	Escala
Clara satisfacción	6	A

Más satisfecho que insatisfecho	1	B
No definido	0	C
Más insatisfecho que satisfecho	0	D
Máxima insatisfacción	0	E
Contradictorio	0	F

Cálculo del Índice de Satisfacción Grupal

ISG = A (+1) + B (+0.5) /N

ISG = (6(+1) + 1(+0.5)) / 7 = (6 + 0.5) / 7

ISG = 0.92

Interpretación del resultado del ISG

El valor obtenido del ISG fue 0.92 lo que indica un buen nivel de satisfacción de los usuarios con respecto a la propuesta solución. Por lo que se puede afirmar que se cumplió con el objetivo de la investigación.

3.5 Pruebas de Aceptación

Las pruebas de aceptación tienen como función validar que el sistema cumpla con el funcionamiento esperado y permitir al usuario de dicho sistema que determine su aceptación, desde el punto de vista de su funcionalidad y rendimiento, estas pruebas las realiza el cliente. Con respeto a las pruebas funcionales, se realizan sobre el sistema completo, y buscan una cobertura de la especificación de requisitos, no se realizan durante el desarrollo, pues sería impresentable para el cliente; son presentadas una vez pasada todas las pruebas de integración por parte del desarrollador. Una prueba de aceptación es como una caja negra, cada una de ellas representa una salida esperada del sistema.

Es responsabilidad del cliente verificar la corrección de las pruebas de aceptación y tomar decisiones acerca de las mismas. Se emplean técnicas denominadas "pruebas alfa" y "pruebas beta". Las pruebas alfa consisten en invitar al cliente a que pruebe el sistema, se trabaja en un entorno controlado y el cliente siempre tiene un experto a mano para ayudarle a usar el sistema y para analizar los resultados. Las pruebas beta vienen después de las pruebas alfa, y se desarrollan en el entorno del cliente, un entorno

que está fuera de control, aquí el cliente se queda solo con el producto y trata de encontrarle fallos (reales o imaginarios) de los que informa al desarrollador [44]. Estos tipos de pruebas proporcionaron como resultado un acta de aceptación (ver anexo 3) del cliente en aprobación general con la solución desarrollada.

3.6 Pruebas de integridad al repositorio de Nova

Con el fin de demostrar el cumplimiento del objetivo planteado y la satisfacción del cliente con el resultado obtenido, se realizaron pruebas para valorar la calidad del módulo implementado. Las imágenes que se muestran a continuación corresponden a fragmentos de pruebas de integridad realizadas al repositorio de Nova:

Figura 5: Prueba de integridad realizada al repositorio de Nova.

(Fuente: Elaboración propia)

3.6.1 Pruebas de integración con *Jenkins*

Las imágenes que a continuación se muestran pertenecen a un conjunto de pruebas realizadas al repositorio de Nova, usando *Jenkins* como interprete para la ejecución de la solución implementada:

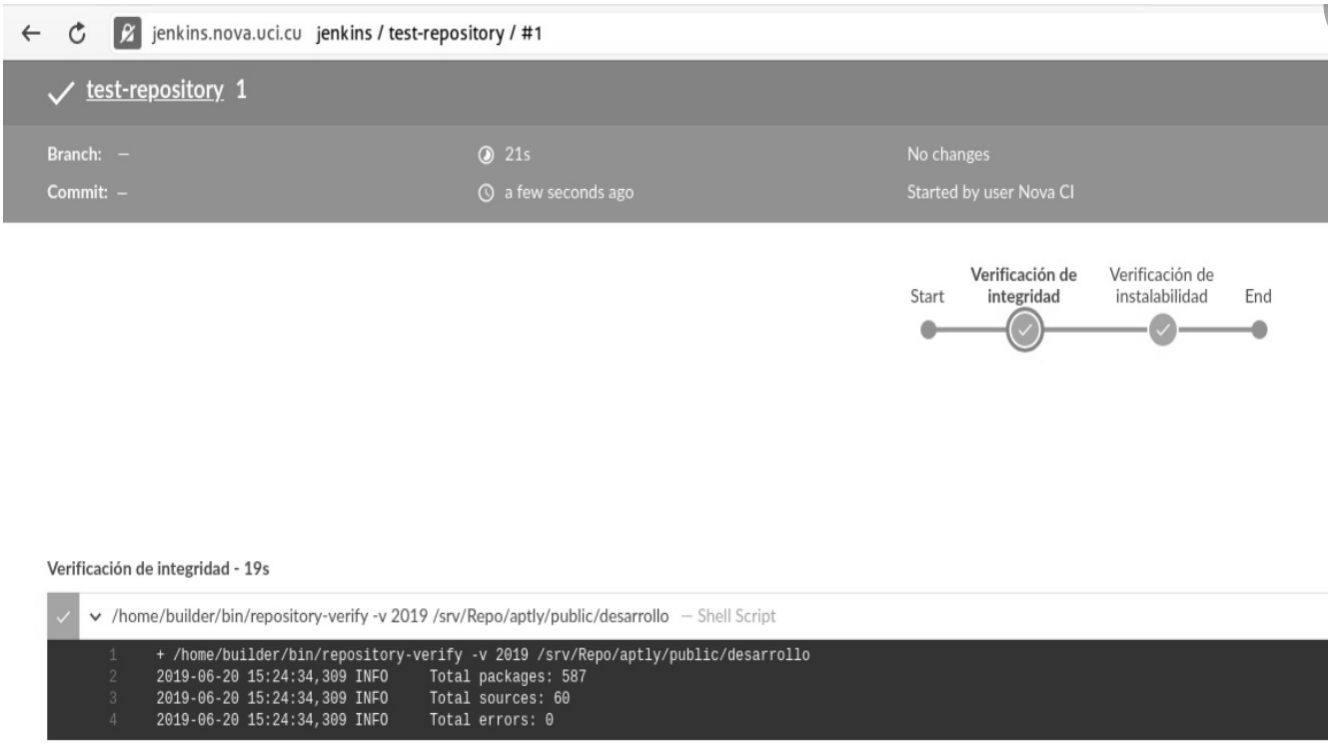


Figura 6: Ejecución de la herramienta para pruebas de integridad al repositorio de nova desde *Jenkins.*

(Fuente: Elaboración propia)

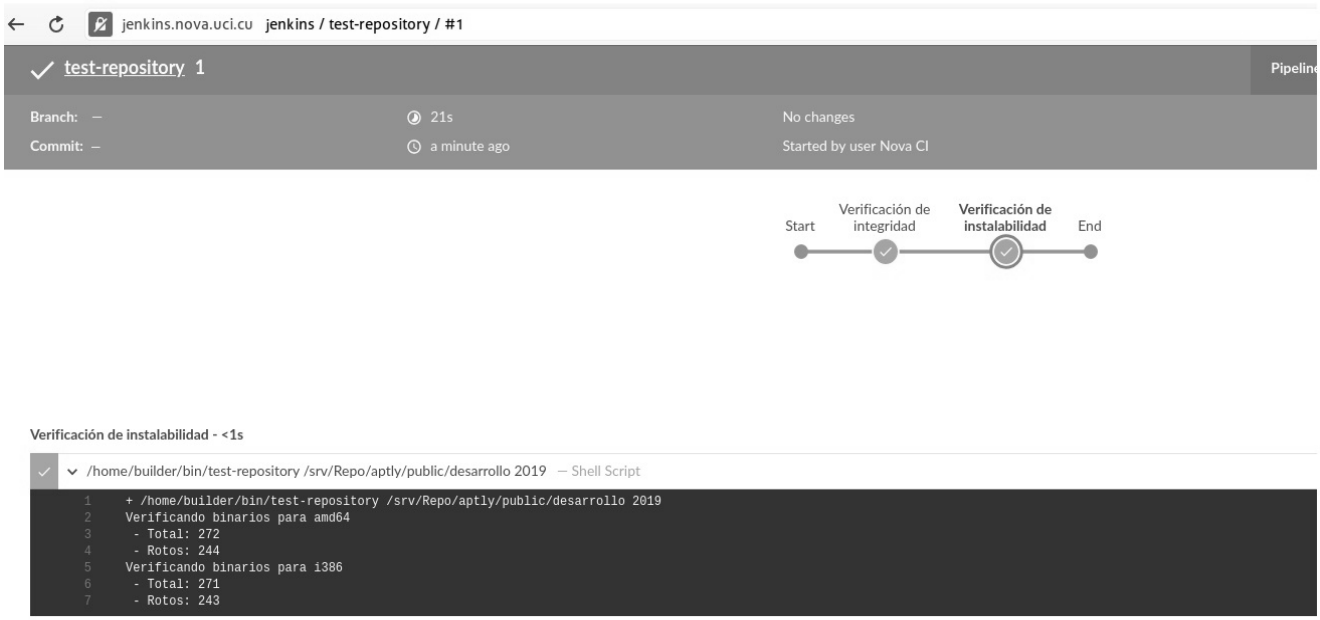


Figura 7: Ejecución de la herramienta para pruebas de integridad al repositorio de Nova desde *Jenkins*. Analizado de constructibilidad e instalabilidad.

(Fuente: Elaboración propia)

3.7 Conclusiones parciales

Al finalizar el desarrollo del presente capítulo se realizó la implementación y ejecución de pruebas correspondiente a la herramienta propuesta, obteniéndose como resultado un sistema completamente funcional. También se mostró parte del código fuente, como mecanismo para ilustrar el proceso de implementación y ejecución del sistema. De igual forma se realizó la descripción del caso de prueba

basado en uno de los requisitos utilizados en las pruebas a la propuesta solución. Para realizar la verificación del sistema se decidió utilizar las técnicas de caja blanca y pruebas de aceptación, dándose mejor acabado al producto a obtener siguiendo los requerimientos del sistema. La ejecución de pruebas realizadas al culminar la implementación de la propuesta solución, garantizó el desarrollo de una herramienta con la calidad requerida por el cliente.

Conclusiones generales

Con la culminación del presente trabajo de diploma se cumplió con cada uno de los objetivos trazados, destacándose de forma general los siguientes aspectos:

1. El estudio de sistemas homólogos para la ejecución de pruebas de integridad en repositorios permitió determinar una solución capaz de detectar inconsistencias, así como los métodos prácticos para la comprobación de integridad en los paquetes del repositorio de Nova.
2. Se desarrolló una herramienta capaz de detectar modificaciones en los paquetes almacenados en el repositorio de la distribución cubana GNU/Linux Nova, logrando detectar debilidades en los paquetes que requerían de un consumo considerable de tiempo.
3. Las pruebas realizadas al producto implementado comprobaron el cumplimiento de los requisitos definidos durante la investigación y las peticiones del cliente.

Recomendaciones

La herramienta de pruebas de integridad al repositorio de Nova dispone de las funcionalidades necesarias para el análisis de los paquetes, no obstante se sugieren las siguientes recomendaciones:

- Añadirle a la herramienta una funcionalidad que verifique las firmas digitales de los paquetes.
- Introducir a la herramienta un mecanismo que pueda determinar el grado de calidad del repositorio en dependencia del total de paquetes almacenados en el repositorio y la cantidad de inconsistencias detectadas.

Bibliografía

1. D'Alos-Moner, Adela. Repositorios digitales: un concepto, múltiples visiones. Disponible en: http://www.doc6.es/media/pdfs/articulos/Repositorios_digitales.pdf . España, 29 de junio de 2009. [Citado el 26 de octubre de 2018].
2. Centro de *software* Libre (CESOL). Disponible en: http://www.uci.cu/investigacion-y-desarrollo/centros-de-desarrollo/centro-de-*software*-libre-cesol. [Citado el 24 de octubre de 2018].
3. Pierra Fuentes, Allan. 20 de octubre de 2011. Trabajo final presentado en opción al título de Máster en Informática Aplicada: Conceptualización y Reestructuración estratégica de la Distribución cubana de GNU/Linux Nova. Universidad de las Ciencias Informáticas, La Habana, Cuba, 20 de octubre de 2011 .
4. Isotton, A. Debian Repository HOWTO. Disponible en: http://www.isotton.com/*software* /debian/docs/repository-howto/repository-howto.html. [Citado el 31 de octubre de 2018].
5. Obregon, A. Repositories - Community Ubuntu Documentation. Repositories. Disponible en:<https://help.ubuntu.com/community/Repositories>. [Citado el: 25 de noviembre de 2015].
6. Gómez, S. Manual para la gestión de *software*: Guía definitiva para la gestión. Disponible en: http://docs.fedoraproject.org/es. [Citado el 4 de diciembre de 2015].
7. ¿Qué es exactamente un Paquete de *software*? | MicroTeknologias, [no date]. Disponible en: https://microteknologias.wordpress.com/2009/03/13/%C2%BFque-es-exactamente-un-paquete-de*software*/. [Citado el 11 Octubre 2015].
8. Schwarz, Ian Jackson and Christian. Debian Policy Manual. Debian Policy Manual, 1998. Disponible en: https://www.debian.org/doc/debianpolicy/#document-index. [Citado el 4 de febrero del 2018].
9. Real Academia Española, Asociación de Academias de la Lengua Española. Diccionario de la lengua española. Disponible en: http://dle.rae.es.
10. Pressman, Roger S. 2010. Ingeniería de *software*. Enfoque práctico. 7ma edición. New York: EUA, 2010. ISBN:978-0-07-337597. [Citado el 12 de noviembre de 2018].

11. López Manrique, Yuri Vladimir. Computación Forense: Una Forma de obtener para combatir y prevenir delitos informáticos. Escuela de Ingeniería en Ciencias y Sistemas, Facultad de Ingeniería, Universidad de San Carlos de Guatemala. Guatemala, marzo de 2007.
12. Langasek, Steve; Ogasawara, Leann; Planella, David; Skaggs, Nicholas. Ubuntu Manpage: apt-cache - query the APT cache. Disponible en: http://manpages.ubuntu.com/manpages/precise/en/man8/apt-cache.8.html. [Citado el 25 Octubre 2015].
13. Langasek, Steve; Ogasawara, Leann; Planella, David; Skaggs, Nicholas.Ubuntu Manpage: apt-rdepends - performs recursive dependency listings similar to. Disponible en: http://manpages.ubuntu.com/manpages/hardy/man8/aptrdepends.8.html. [Citado el 25 Octubre 2015].
14. García Dominguez, A. Reprepro. Disponible en: https://mirrorer.alioth.debian.org/. [Citado el: 16 Octubre 2015].
15. DuraSpace Registry - Duraspace.org. Duraspace.org. [Citado el 16 de octubre de 2018].
16. De Giusti, Marisa. Oviedo, Nestor. Lira, Ariel. Luján Villareal, Gonzalo. Control de integridad y calidad en repositorios DSpace. Ponencia. III Conferencia Bibliotecas y Repositorios Digitales de América Latina (BIREDIAL '13). VIII Simposio Internacional de Bibliotecas Digitales (SIBD '13) "ACCESO ABIERTO, PRESERVACIÓN DIGITAL Y DATOS CIENTÍFICOS". Ciudad de la Investigación, Universidad de Costa Rica, Costa Rica, 17 de octubre de 2013.
17. Koji. Disponible en: https://fedoraproject.org/wiki/Koji. [Citado el 5 de diciembre de 2018].
18. Collins-Sussman, Ben. Michael Pilato, C. Control de versiones con Subversion. Capítulo 5. Administración del Repositorio. Mantenimiento del Repositorio. Diponible en: http://svnbook.red-bean.com/es/1.0/svn-ch-5-sect-3.html. [Citado el 5 de diciembre de 2018].
19. Smirnov, Andrey. APTLY. Disponible en: https://www.aptly.info. [Citado el 28 de mayo del 2019].
20. Langasek, Steve; Ogasawara, Leann; Planella, David; Skaggs, Nicholas. Ubuntu manuals. Ceve – parse package metadata. Disponible en:

    https://manpages.ubuntu.com/manpages/precise/man1/ceve.1.htlm#options. [Citado el 16 de octubre

del 2018].

21. Kili, Aaron. Cómo comprobar la suma md5 de los paquetes que tenemos instalados en nuestro sistema basado en Debian o Ubuntu. Disponible en: https://www.wifi-libre.com/topic-584-comprobar-la-integridad-de-los-paquetes-deb-con-debsums.html. [Citado el 6 de diciembre de 2018].
22. Tripwire. Disponible en: http://www.linux-xd.com.ar/manuales/rh9.0/rhl-rg-es-0/ch9/ch-tripwire.htlmtripwire.html. [Citado el 6 de diciembre del 2018].
23. Abate, Pietro; Vouillon, Jerome. Disponible en: http://www.mancoosi.org. Universidad de París-Diderot, París, Francia, 3 de enero de 2016. [Citado el 23 de mayo del 2018].
24. Sosa Fridrij, Liuba. Mecanismo para control de Integridad de las bibliotecas dinámicas del sistema SCADA Guardián del Alba. Trabajo de diploma para optar por el título de Ingeniera en Ciencias Informáticas. Universidad de las Ciencias Informáticas, La Habana, Cuba, junio de 2013.
25. Sánchez Rodríguez, Támara. Metodología de desarrollo para la actividad productiva de la UCI, 3 de junio de 2015. [Citado el 7 de noviembre del 2018].
26. Jean-François Pillou, Jeff. Lenguajes de Programación. Grupo Figaro, CMM Benchmark. Francia, 2018. Disponible en: https://es.ccm.net/contents/304-lenguajes-de-programacion. [Citado el 7 de noviembre del 2018].
27. Stallman, Richard. (forwarded with comments by Chet Ramey) (10 de febrero de 1988). GNU + BSD = ?. (Google Groups). URL accedida el 22 de marzo de 2011.
28. Contreras Vasallo, Cecilia; De la Guarda Reyes, Alfonso; Vassallo Rubiños, Ana María. A Byte of Python. Características de Python. Disponible en: http://dev.laptop.org/~edsiper/byteofpython_spanish/ch01s02.html. [Citado el 6 de marzo de 2019].
29. Hernández Orallo, Enrique. El lenguaje Unificado de Modelado (UML). [Citado el 8 de noviembre del 2018].
30. VISUAL PARADIGM FOR UML. Libere las revisiones de la transferencia directa y del *software* |CNET. [En línea]. Disponible en: http://descargar.cnet.com/Visual-Paradigm-for-UML/3000-2247_4-

42700.html. [Citado el 28 de noviembre del 2017].

31. Anderson, Nate; Fisher, Ken. Visual Studio Code editor hits version 1, has half a million users. Ars Technica. Condé Nast. 15 de abril de 2016.
32. Béjar, Rubén. J Lopez-Pellicer, Francisco. Latre, Miguel A. Nogueras-Iso, Javier. Zarazaga-Soria, Javier. Github como herramienta docente. Departamento de Informática e Ingeniería de Sistemas, Universidad de Zaragoza, España, 2015.
33. García Oterino, Ana. ¿Qué es Jenkins? 9 de mayo de 2014. Disponible en: https://www.javiergarzas.com/2014/05/jenkins-en-menos-de-10-min.html. [Citado el 11 de abril de 2019].
34. Rome, José Enrique. Integración y Despliegue Continuo: Monitorización. Sevilla, 2016. [Citado el 11 de abril de 2019].
35. Pressman, Roger S. 2010. Ingeniería de *software*. Enfoque práctico. 7ma edición. New York: EUA, 2010. ISBN:978-0-07-337597. [Citado el 1 de marzo de 2019].
36. Ayala Lopez, Angie Natalia; Bonilla Garcia, Sayuri; Niño Rivera, Thalia Xilema; Posada Vivas, Juan Pablo; Rodriguez Cortez, Kimberly. Requerimientos Funcionales y No Funcionales (RF/RNF). Disponible en: http://ingenieriade*software*.bligoo.com.mx/requerimientosfuncionales-y-no-funcionales-rf-rnf. [Citado el 12 de febrero de 2019].
37. ISO 25000 Calidad de producto de software. La familia de normas ISO/IEC 25000, 2019. Disponible en: https://iso25000.com/index.php/normas-iso-25000?limit=4&start=4. [Citado el 1 de abril del 2019].
38. Sommerville, Ian. Ingeniería de *software*. Pearson Addison Wesley, Madrid, España, 2005. ISBN:84-7829-074-5. [Citado el 12 de febrero de 2019].
39. Jeffries, Ron; Anderson, Ann; Hendri, Chet. Extreme Programming Installed. s.l.: Addison Wesley Professional, 2001. [Citado el 8 de marzo del 2018].
40. Nava, Maily. Estilos Arquitectónicos. Arquitectura de software. Sistemas de llamada y retorno. Disponible en: <http://es.scribd.com/doc/23161581/Estilos-Arquitectonico>. [Citado el 15 de marzo del

2013].

41. Arias Calleja, Manuel. Estándares de Codificación. España, 2015.

    Disponible en: https://docplayer.es/11247774-Carmen-estandares-de-codificacion-manuel-arias-calleja.html. [Citado el 1 de abril del 2019].

42. Arias Calleja, Manuel. Carmen: una herramienta de software libre para modelos gráficos probabilistas. España, 2009. [Citado el 1 de abril del 2019].

43. Fernández de Castro, F. A. Análisis de la medición del impacto en los proyectos de investigación de la Universidad Agraria de La Habana (UNAH), 100pp., Tesis en opción al título de Máster en Desarrollo Agrario y Rural, Universidad Agraria de La Habana, La Habana, Cuba, 2010.

44. Zapata Sánchez, Javier. Pruebas de *Software*. Ingeniería de *Software* con énfasis en pruebas. Niveles de Prueba del *Software.* Disponible en: https://pruebasdelsoftware.wordpress.com/. Bogotá, Colombia, 2013. [Citado el 2 abril del 2019].

Anexo 1

Entrevista realizada a varios miembros del centro CESOL, con el objetivo de desarrollar la introducción del presente documento e indagar sobre la situación problemática:

¿Qué es la integridad de un paquete?

¿Cómo se realiza el proceso de pruebas de integridad al repositorio de Nova en el centro?

¿Qué opina sobre de los beneficios que tendría para el centro disponer de una herramienta que automatice el proceso de pruebas de integridad para el repositorio de Nova?

Anexo 2

Encuesta realizada a los desarrolladores de CESOL correspondiente a la técnica de IADOV:

Se le insta a responder el siguiente cuestionario con el fin de conseguir su colaboración en la presente investigación; por favor exprese un criterio que guíe al autor del trabajo. Marque con una X en las preguntas donde se brinda una opción a escoger y en los casos contrarios responda brevemente.

1. ¿Piensa necesario que se verifique la integridad en los paquetes del repositorio de la distribución cubana GNU/Linux Nova?

   Sí____ No____ No sé____

2. ¿Considera usted correcta la forma en que se realiza el proceso de verificación de la integridad en los paquetes del repositorio de la distribución cubana GNU/Linux Nova?

   Sí____ No____ No sé____

3. ¿Considera factible contar con una aplicación informática que permita detectar los paquetes averiados ?

   Sí____ No____ No sé____

4. Luego de haber visto la automatización del proceso de verificación de la integridad en los paquetes del repositorio de la distribución cubana GNU/Linux Nova, represente en que nivel le gusta la solución desarrollada

   ____Me gusta mucho ____Me disgusta mas de lo que me gusta

   ____Me gusta mas de lo que me disgusta ____No me gusta nada

   ____Me da lo mismo ____No se decir

5. ¿Qué opina usted sobre los beneficios que traería para el centro CESOL disponer de una solución que automatice el proceso de verificación de la integridad en los paquetes del repositorio de la distribución cubana GNU/Linux Nova?

Anexo 3

UCi Universidad de las Ciencias Informáticas

Acta de aceptación de productos de trabajo

ACTA DE ACEPTACIÓN DE PRODUCTOS DE TRABAJO

En cumplimiento del **Convenio de colaboración** establecido entre el **Centro de Software Libre (CESOL)** y el estudiante **Pedro Javier Hernández Melgarejo** de la Facultad 1 de la Universidad de las Ciencias Informáticas y en función de la ejecución del proyecto: **Herramienta de pruebas de integridad al repositorio de Nova**, se hace entrega del producto que se relaciona a continuación:

- Herramienta de pruebas de integridad al repositorio de Nova.

La parte Cliente, luego de haber revisado el producto de trabajo relacionado anteriormente procede a firmar la aceptación de los mismos en total conformidad.

Entrega	**Recibe**
Nombre y apellidos: Pedro Javier Hernández Melgarejo	**Nombre y apellidos:** Yurisbel Vega Ortiz
Cargo: Estudiante Facultad 1	**Cargo:** Director de CESOL
Firma:	**Firma:**

Fecha: 31/05/2017