# Integración continua en el proceso de construcción y publicación de paquetes para la Distribución Cubana de GNU/Linux Nova

## Trabajo de Diploma para optar por el título de Ingeniero en Ciencias Informáticas

Autor: Manuel Alejandro Ricardo Serrano

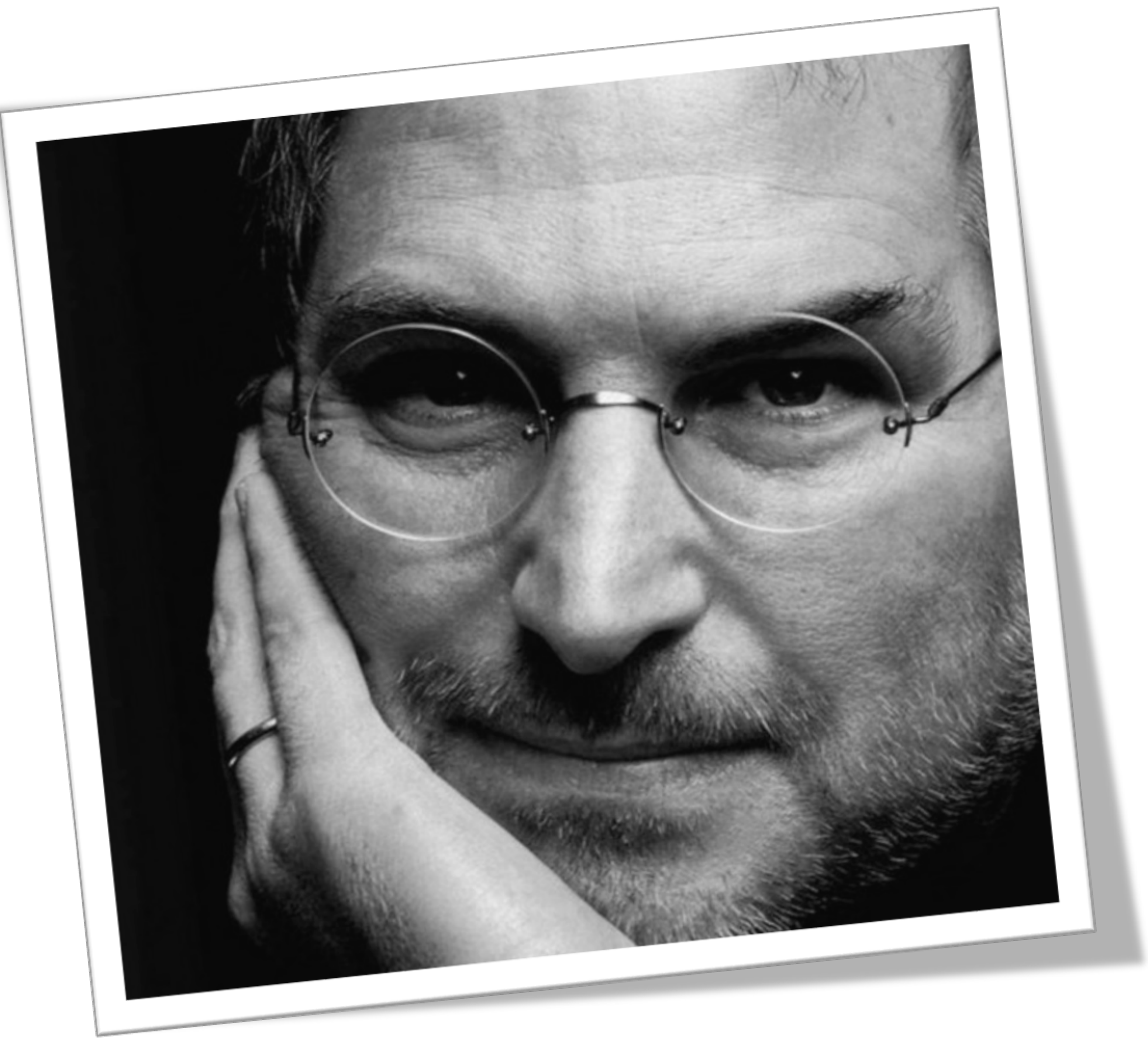
Tutor: Ing. Juan Manuel Fuentes

Ing. Yileni Hechavarría González

Consultante: Ing. Javier Piñeiro Cárdenas

**Universidad de las Ciencias Informáticas**

**La Habana, Cuba. Junio, 2018.**



*"…la única manera de hacer un trabajo genial, es amar lo que haces…"*

***Steve Jobs.***

## 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.

**Manuel Alejandro Ricardo Serrano**

__________________

Firma de la autor

| **Ing. Juan Manuel Fuentes Rodríguez** | **Ing. Yileni Hechavarría González** |
| --- | --- |
| Firma del tutor | Firma de la tutor |

## Dedicatoria

A mis padres, Milagro Serrano Liens y Bárbaro Ricardo Hidalgo les dedico el resultado final de toda mi carrera como estudiante. Por estar en cada momento y darme apoyo incondicional toda la vida para que sea una mejor persona, por darme todo el amor y a su vez ser exigentes para que lograra ser un profesional. Gracias, hoy y siempre gracias, este gran logro es por ustedes. También dedico mi logro a mi Suyi, que año tras año en la carrera estuvo siempre en las buenas y en las malas, sin dejarme ni un sólo segundo sin su ayuda. A mi hijo que espero, para que algún día esté orgulloso de papá y siga mis pasos . A mi familia, que de una forma u otra han sido siempre parte de mi éxito, a mis amigos de las mil batallas, esos amigos de las largas noches de estudio codo a codo, a esos Titanes que han luchado junto a mi por ser un día todos ingenieros, a Raúl, Mauro, Amado, Michel, Juan Carlos, Carlitos, Ramón, Vlado, Osvaldo, a mi grupo 03 y a todos mis restantes amigos de la UCI y a los de afuera también, todos ustedes son parte de mi éxito.

## Agradecimientos

Eternamente agradecido a mis tutores Yileni y Juan Manuel, a mi tutor consultado Javier, gracias a ustedes he podido realizar un trabajo satisfactorio. Haberlos tenido como guías para desarrollar cada paso de mi aplicación y documento de tesis, ha sido un placer y es un orgullo haber trabajado con ingenieros como ustedes.  A cada profesor de la UCI que fue parte mi formación, a cada amigo que me apoyó y a los especialistas e ingenieros del centro CESOL, también a toda mi familia, a mis padres y a mi novia. Viviré siempre muy agradecido con todos.

## Resumen

En la distribución cubana GNU/Linux Nova actualmente se almacena todo el código fuente de los paquetes en un repositorio. Para monitorizar el versionado de los paquetes el proyecto utiliza como controlador de versiones la herramienta Git. Una vez terminado el proceso de empaquetamiento de un paquete, se le entrega a la persona encargada de subir los paquetes al repositorio de forma manual, corriendo el riesgo de perder información o introducir errores a la hora de integrarlos al mismo, además, se publican los paquetes sin ser firmados por la llave primaria previamente. El objetivo del presente trabajo es elevar el nivel de seguridad, automatizando el proceso de construcción y publicación de paquetes de código fuente y binario, mediante integración continua para la distribución cubana de GNU/Linux Nova. Para guiar el proceso de construcción de la propuesta de solución se utiliza la metodología de desarrollo de software AUP variación para la UCI y Jenkins como herramienta de integración continua.

**Palabras clave:** integración continua, paquete, seguridad, repositorio.

# Índice

Introducción ..... 1
Capítulo 1: Fundamentación teórica ..... 5
1.1 Introducción ..... 5
1.2 Marco conceptual ..... 5
1.2.1 Paquete en GNU/Linux ..... 5
Paquetes fuentes ..... 5
Paquetes binarios ..... 6
1.2.2 Dependencia de paquetes ..... 7
1.2.3 Empaquetado ..... 8
1.2.4 Repositorio de paquetes ..... 9
1.2.5 Sistema de control de versiones ..... 9
1.3 Proceso de Integración continua ..... 9
Entrega continua y Despliegue continuo ..... 12
1.4 Análisis de herramientas de Integración continua ..... 13
1.4.1 Herramienta de Integración continua (*Jenkins*) ..... 18
1.5 Metodología para guiar el desarrollo de la solución ..... 21
1.5.1 AUP variación para la UCI ..... 21
1.6 Lenguajes de programación y herramientas a utilizar ..... 22
1.6.1 Lenguajes de programación ..... 22
1.6.2 Sistema de control de versiones Git ..... 23
Estándar DEP 14 ..... 25
1.6.3 Dpkg ..... 26
1.6.4 Sbuild ..... 26
1.6.5 Gbp ..... 26
1.6.6 Visual Paradigm ..... 27
1.6.7 Aptly ..... 27
1.6.8 Piuparts ..... 27

1.6.9 Lintian ........ 28
1.7 Conclusiones parciales ........ 28
Capítulo 2: Concepción de la propuesta de solución ........ 29
2.1 Introducción ........ 29
2.2 Modelo de dominio ........ 29
2.3 Condiciones para el funcionamiento de la solución ........ 30
2.4 Requisitos del sistema ........ 30
2.4.1 Definición de las técnicas de obtención de requisitos ........ 30
2.4.2 Requisitos funcionales ........ 31
2.4.3 Requisitos no funcionales. ........ 31
2.4.4 Validación de los requisitos ........ 32
2.5 Historias de usuarios ........ 33
2.6 Definición de la arquitectura ........ 37
2.6.1 Patrón arquitectónico ........ 37
2.7 Descripción de la propuesta de solución ........ 38
2.8 Conclusiones parciales ........ 39
Capítulo 3: Construcción y validación de la propuesta de solución ........ 40
3.1 Introducción ........ 40
3.2 Diagrama de despliegue ........ 40
3.3 Estándares de codificación ........ 41
3.4 Pruebas de *software* ........ 43
3.4.1 Pruebas de internas ........ 43
3.4.2 Pruebas de Aceptación ........ 46
3.5 Técnica de índice de satisfacción grupal (IADOV) ........ 49
3.6 Resultados de la investigación ........ 52
3.7 Conclusiones parciales ........ 52
Conclusiones generales ........ 54
Recomendaciones ........ 55
Bibliografía ........ 56

Anexo 1 ..... 60
Anexo 2 ..... 61
Anexo 3 ..... 62

# Índice de Figuras

Figura 1: Componentes de un sistema de integración continua. ....................11
Figura 2: Proceso de Integración continua. ....................13
Figura 3: Porcentaje de uso de herramientas de integración continua a nivel mundial. ....................14
Figura 4: Ciclo de vida del pipeline. ....................21
Figura 5: Flujo de trabajo básico en Git. ....................24
Figura 6: Modelo de dominio ....................29
Figura 7: Patrón arquitectónico Tuberías y filtros. ....................37
Figura 8: Flujo del proceso de Integración continua. ....................39
Figura 9: Diagrama de despliegue ....................41
Figura 10: Estándares de codificación ....................42
Figura 11: Procedimiento de la etapa publicación de binarios. ....................44
Figura 12: Grafo de flujo y cálculo de caminos linealmente independientes. ....................45
Figura 13: Resultado de las pruebas. ....................47
Figura 14: Ubicación del índice de satisfacción grupal. ....................51

# Índice de tablas

Tabla 1: Las mejores 10 herramientas de integración continua según comunidades Slant. ......................15
Tabla 2: Las mejores 5 herramientas de integración continua para código abierto. ..................................15
Tabla 3: Comparación GitLab CI y Jenkins. ..........................................................................................16
Tabla 4: Historia de usuario #1. .............................................................................................................33
Tabla 5: Historia de usuario #2. .............................................................................................................34
Tabla 6: Historia de usuario #3. .............................................................................................................35
Tabla 7: Historia de usuario # 5. ............................................................................................................35
Tabla 8: Historia de usuario # 6. ............................................................................................................36
Tabla 9: Historia de usuario # 7. ............................................................................................................36
Tabla 10: Caso de prueba para el camino 1. ..........................................................................................45
Tabla 11: Caso de prueba para el camino 2. ..........................................................................................46
Tabla 12: Caso de prueba para el camino 3. ..........................................................................................46
Tabla 13: Caso de prueba para el camino 3. ..........................................................................................49
Tabla 14: Escala de satisfacción. ...........................................................................................................50

## Introducción

El desarrollo del *software* se mantiene en una constante evolución producto a la necesidad de satisfacer los problemas de una sociedad informatizada. Cuba, a pesar de ser un país subdesarrollado cuenta con instituciones capacitadas para los posibles cambios tecnológicos que esto provoca. La Universidad de las Ciencias Informáticas (UCI) es una de ellas: creada con el objetivo de formar profesionales altamente calificados en la rama de la Informática, encargados de producir aplicaciones y brindar servicios de soporte a la industria cubana del *software* [1].

La migración a *software* libre constituye una de las necesidades a resolver en la informatización de la sociedad cubana, por lo cual la UCI en función de dicha necesidad creó un centro dedicado al desarrollo de soluciones de código abierto llamado Centro de S*oftware* Libre (CESOL), encargado de desarrollar una distribución de GNU/Linux autóctona capaz de facilitar el trabajo cotidiano de los usuarios, un sistema libre de licencias privativas que puede ser utilizado para la migración a estándares abiertos en los Órganos y Organismos de la Administración Central del Estado: la Distribución Cubana de GNU/Linux Nova.

Nova posee varios productos que hacen que la distribución sea sostenible, socio-adaptable, segura y soberana tecnológicamente. Para que cuenten los productos con estas características, utiliza el núcleo de Linux e incluye determinados paquetes de aplicaciones informáticas [2]. En Nova los proyectos son organizados y guardados en un repositorio nacional donde se brindan actualmente miles de paquetes, en varios lenguajes de programación. La gestión de los paquetes en el repositorio representa un reto para el colectivo joven con poca experiencia que desarrolla la distribución de GNU/Linux Nova, pues debe existir un dominio del conocimiento del desarrollo de las versiones anteriores para dar continuidad a los productos y así reutilizar el trabajo anterior para economizar tiempo.

Cuando se necesita modificar o realizar pruebas en el código fuente de un programa, los desarrolladores descargan el paquete de código fuente del repositorio y lo descomprimen, luego crean parches para ser aplicados al paquete, provocando que el código fuente pueda ser sometido a modificaciones mediante los propios parches o a pruebas de errores, posteriormente lo empaquetan. Una vez terminado exitosamente

el proceso de empaquetamiento se le entrega a la persona encargada de subir los paquetes al repositorio, por lo que se corre el riesgo de perder información durante el traslado. Esta fase del proceso se realiza de forma manual quedando expuesto a posibles pérdidas de código fuente o errores a la hora de integrar los paquetes al repositorio, además, se publican los paquetes sin ser firmados por la llave primaria previamente.

A partir de la situación problemática existente se plantea el siguiente **problema de investigación:** ¿cómo elevar el nivel de seguridad en el proceso de construcción y publicación de paquetes de *software* para la Distribución Cubana de GNU/Linux Nova? Para dar solución al problema en cuestión se propone como **objeto de estudio:** el proceso de construcción y publicación de paquetes de código fuente y binario, en consecuencia, se centra el **campo de acción** en el proceso de construcción y publicación de paquetes de código fuente y binario en la Distribución Cubana de GNU/Linux Nova.

Por lo que se formula como **objetivo general:** elevar el nivel de seguridad automatizando el proceso de construcción y publicación de paquetes de código fuente y binario mediante integración continua para la Distribución Cubana de GNU/Linux Nova. A partir del objetivo general planteado se derivan los siguientes **objetivos específicos:**

- Elaborar el marco teórico de la investigación sobre el proceso de construcción y publicación de paquetes para la Distribución Cubana de GNU/Linux Nova.
- Implementar configuraciones que permitan automatizar el proceso de construcción y publicación de paquetes para la Distribución Cubana de GNU/Linux Nova.
- Evaluar el correcto funcionamiento de la solución.

Como **preguntas científicas** se plantean:

- ¿Cuáles son las tendencias actuales relacionadas con las herramientas de integración continua?
- ¿Qué tecnologías, herramientas y metodología se requieren utilizar para automatizar el proceso de construcción y publicación de paquetes de código fuente y binario mediante integración continua?

- ¿Qué resultado se obtendrá al validar la automatización del proceso de construcción y publicación de paquetes de código fuente y binario mediante integración continua?

Para dar solución al problema planteado se definieron las siguientes **tareas de investigación:**

- Definición de los conceptos relacionados con el marco teórico de la investigación y las tecnologías necesarias para el desarrollo en el proceso de construcción y publicación de paquetes fuentes y binarios.
- Estudio de soluciones existentes en el proceso de construcción y publicación de paquetes para lograr un mejor entendimiento del objeto de estudio y el campo de acción.
- Estudio de lenguajes y herramientas para el desarrollo de la solución.
- Identificación de los requisitos para elaborar la propuesta de solución.
- Validación de los requisitos del sistema aplicando la técnica de diseño de casos de prueba.
- Desarrollo de las pruebas de *software* diseñadas para medir la calidad de la solución.

Para desarrollar esta investigación se utilizan varios métodos:

**Histórico-Lógico**: permitió conocer la evolución que ha tenido el proceso de desarrollo en cuanto al uso de paquetes en la Distribución Cubana de GNU/Linux Nova, así como las causas que dieron paso a su surgimiento, las características comunes que se han mantenido con el transcurso del tiempo y las nuevas que han surgido.

**Analítico-Sintético:** con el fin de descomponer el problema de investigación en elementos por separado (identificación de requisitos y repositorios de códigos fuente y binarios) y profundizar en el estudio de cada uno de ellos, para luego sintetizarlos en la solución propuesta.

**Entrevista:** utilizada para conocer cómo se realiza el proceso de construcción y publicación de paquetes en la Distribución Cubana de GNU/Linux Nova, aplicada a los especialistas del centro.

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:** se tratan los conceptos y aspectos más significativos abordados en diferentes fuentes bibliográficas, que se relacionan con el proceso de construcción y publicación de 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. Concepción de la propuesta de solución**: se define la propuesta de solución y la arquitectura y el patrón arquitectónico a utilizar. Se obtienen los requisitos funcionales, no funcionales y se describen mediante el uso de historias de usuarios.

**Capítulo 3. Construcción y validación de la propuesta de solución**: se definen los estándares de codificación para desarrollar una programación homogénea y 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

En el presente capítulo se realiza un estudio de los conceptos y aspectos más significativos que se relacionan con el proceso de construcción y publicación de paquetes fuentes y binarios en la Distribución Cubana de GNU/Linux Nova. Son consultadas diferentes fuentes bibliográficas para profundizar y ampliar el conocimiento referente a los lenguajes y herramientas para la gestión de los paquetes. Se realiza un análisis a diversos sistemas homólogos para facilitar la selección de la mejor herramienta para el desarrollo de la solución.

## 1.2 Marco conceptual

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

### 1.2.1 Paquete en GNU/Linux

Un paquete de *software* en una distribución GNU/Linux es generalmente un archivo comprimido que posee una estructura interna predefinida, que permite ser manipulado por herramientas de gestión de *software* (Gestores de Paquetes) para lograr su compilación o instalación, actualización y eliminación en las distribuciones, de forma cómoda, segura, estable y centralizada [3]. Se pueden diferenciar en los siguientes tipos de paquetes:

**Paquetes fuentes**

Los paquetes fuentes contienen aplicaciones libres o de código abierto, incluyen el código fuente de las mismas y el resto de los recursos que se requieran para su compilación. En el caso de las distribuciones basadas en Debian GNU/Linux, los paquetes fuentes conforman un grupo de archivos que contienen las modificaciones, parches del mantenedor o desarrollador de Debian. Además, se provee un archivo de descripción que contiene información sobre el conjunto de archivos que conforman al paquete fuente, así como firmas de verificación de integridad de los mismos, datos del mantenedor y para qué versión de la

distribución de GNU/Linux están empaquetados [4].

Tipos de archivos [5]:

- . dsc : archivo de descripción del paquete (información sobre el paquete).
- . orig.tar.gz : archivo fuente original.
- .diff.gz : archivo con los cambios de Debian sobre el código fuente original.

Dentro de los paquetes fuentes se pueden encontrar los siguientes formatos:

- **Formato 1.0:** consiste en un .orig.tar.gz asociado a un .diff.gz o un solo .tar.gz (en ese caso se dice que el paquete es nativo). Crear un paquete nativo es crear un *tarball* único con el directorio de origen y crear un paquete no nativo implica extraer el archivo tar original en un directorio ".orig" separado y regenerar el .diff.gz comparando el directorio del paquete fuente con el directorio .orig.
- **Formato 3.0 (nativo):** es una extensión del formato del paquete nativo como se define en el formato 1.0. Es compatible con todos los métodos de compresión e ignorará de manera predeterminada cualquier archivo y directorio específico de un sistema de control de versiones.
- **Formato 3.0 (quilt)**: contiene al menos un *tarball* original (.orig.tar.ext) donde la extensión .ext puede ser de tipo .gz, .bz2 y .lzma y un *tarball* de Debian de tipo .debian.tar.ext. También puede contener archivos .tar originales adicionales de tipo .orig-component.tar.ext.

**Paquetes binarios**

Son paquetes en los cuales los componentes que contiene están listos para instalarse en la distribución, poseen una estructura optimizada para un grupo de características de *hardware*, como la arquitectura, generalmente i386[1] o amd64[2], por esta razón cada *hardware* requiere su propio paquete binario. Los

---

1 I386: arquitectura con un procesador de 16 bits con un sistema de memoria segmentada, que permite implementar sistemas operativos con memoria virtual.

2 Amd64: arquitectura desarrollada originalmente por AMD a partir de la arquitectura x86.

paquetes binarios contienen ejecutables, archivos de configuración, páginas de información y otras documentaciones. A continuación, se mencionan los diferentes tipos de paquetes binarios [5]:

- **DEB**: la distribución de Debian se basa en un sistema de gestión de paquetes llamado dpkg[3], donde todos los paquetes se deben proporcionar en el formato de archivo .deb. Contiene dos conjuntos de archivos: uno para instalar en el sistema cuando se instala el paquete y otro que proporciona metadatos adicionales sobre el paquete o se ejecuta cuando se instala o elimina. El segundo conjunto de archivos se denomina archivos de información de control, donde se encuentran los *scripts* y el control del desarrollador del paquete. Se clasifican en los diferentes tipos de archivos:
    - Debian_binary: contiene la versión del archivo .deb
    - Sección de control del paquete (control.tar.gz).
    - data.tar.gz : contiene todos los archivos que se instalarán, con sus rutas de destino.
- **RPM:** por sus siglas en inglés *Redhat Package Manager*, es un tipo de paquetería para Linux desarrollado para la distribución de *Red Hat*, con el fin de construir un sistema fácil de crear e instalar. Una poderosa ventaja sobre otros, es su forma de actualización de las aplicaciones, no necesitan tener los mismos datos que el instalador original, solamente puede incluir los archivos que se actualizarán, esto reduce altamente el peso de sus paquetes.
- ***Ebuild*:** paquete usado solo por la distribución *Gentoo Linux*, consiste en un *script bash* ejecutable en un entorno específico para su distribución. Es una forma automática de compilar e instalar *software*, sus archivos deben tener extensión *.ebuild*.

### 1.2.2 Dependencia de paquetes

En el campo del *software* existe una dependencia de paquetes cuando un paquete requiera de otros para poder funcionar correctamente. A continuación, se mencionan los diferentes tipos de dependencias:

3 **Dpkg:** herramienta para instalar, quitar, y proporcionar información sobre los paquetes .deb.

- Dependencias de construcción (*build dependency*), son aquellos paquetes binarios que se requiere que estén instalados para llevar a cabo el proceso de construcción de un paquete binario a partir de un paquete fuente.
- Dependencias de tiempo de ejecución (*runtime dependency*), son aquellos paquetes que se requieren que estén instalados en el sistema operativo para que determinada aplicación, una vez instalada, pueda ejecutarse [4].

Las dependencias se pueden clasificar en:

- *Depends*: tiene efecto solo cuando se configura un paquete y declara una dependencia absoluta. Un paquete no se configurará a menos que todos los paquetes enumerados en su campo *Depends* hayan sido configurados correctamente. Impone requisitos en el orden en que se configuran, generalmente en una ejecución de instalación, se desempaquetan primero y todos se configuran más tarde [5].
- *Recommends*: declara una fuerte, pero no absoluta dependencia. El campo *Recommends* debe enumerar los paquetes que se recomiendan instalar junto con el paquete que se esté instalando, recomienda hasta las instalaciones menos inusuales [5].
- *Suggests:* declarar que un paquete puede ser más útil con uno o más paquetes. *Suggests* sugiere al sistema de construcción y al usuario que los paquetes enumerados están relacionados con el paquete que se está instalando y que tal vez mejoren su utilidad, pero que instalar este sin ellos es perfectamente razonable [5].

### 1.2.3 Empaquetado

Proceso mediante el cual el mantenedor o desarrollador de una distribución de GNU/Linux, prepara el código fuente de una aplicación, con herramientas que requieren un conjunto de archivos de reglas y datos que especifican cómo se compila (en caso de que lo requiera) y se convierte en un paquete binario listo para ser instalado. El código fuente y el conjunto de archivos de reglas y datos son comprimidos

siguiendo un formato estándar, entendido por las herramientas de creación de paquetes binarios [4].

### 1.2.4 Repositorio de paquetes

Un repositorio de paquetes es el espacio donde los paquetes son almacenados y mantenidos centralmente. Puede ser local o remoto, dependiendo si se encuentra en el sistema de archivos de la máquina del usuario que lo utiliza o si se encuentra en una localización remota accesible a través de la red. Los repositorios de paquetes generalmente cuentan con archivos índices que permiten su localización y la rápida obtención de la información del paquete sin tener que descargarlo o procesarlo. Es común encontrar que los repositorios de las distribuciones que utilizan el paquete binario de Debian una estructura de árbol donde los paquetes se agrupen en directorios del sistema de archivos [4].

### 1.2.5 Sistema de control de versiones

El sistema de control de versiones registra los cambios realizados en un archivo o conjunto de archivos a lo largo del tiempo, de modo que se pueda recuperar versiones específicas más adelante. Permite regresar a versiones anteriores de los archivos del proyecto completo, comparar cambios a lo largo del tiempo, observar las modificaciones, detectar errores, así como ver quién introdujo un problema y cuándo [6]. Para lograr un buen control y monitoreo en la evolución de sus productos, en CESOL se utiliza el controlador de versiones Git, herramienta rápida y eficiente para grandes proyectos.

## 1.3 Proceso de Integración continua

Según su fundador Martin Fowler “La integración continua es una práctica de desarrollo de *software* donde los miembros de un equipo integran su trabajo frecuentemente, como mínimo de forma diaria. Donde cada integración se verifica mediante una herramienta de construcción automática para detectar errores tan pronto como sea posible” [7]. Mejora la calidad en el desarrollo del proceso de manera que se conozcan las fases por las que va pasando el código y el estado del *software* en cada momento. Permite la realización de distintos tipos de pruebas de forma automática, mejora las prácticas de programación y brinda una mayor confianza a la hora de desarrollar en el proyecto [8].

Actividades para llevar a cabo la integración continua [7]:

- Mantener un único repositorio de artefactos.
- Automatizar el proceso de construcción.
- Incluir la automatización de pruebas en el proceso de construcción.
- Enviar cada día al repositorio las últimas versiones de artefactos.
- Por cada envío de códigos se deberá construir la línea base del sistema.
- Mantener el proceso de construcción rápido.
- Probar en un ambiente de pruebas igual al de producción.
- Hacer de una manera fácil que cualquier persona obtenga la última versión del ejecutable.
- Automatizar el proceso de despliegue.

Prácticas de la Integración continua:

- Sincronizar y subir código frecuentemente al menos una vez al día.
- No subir código fuente al repositorio de control de versiones cuando no se puede compilar o posea fallos de las pruebas.
- Reparar los problemas que se presenten a lo largo del desarrollo o corregirse de inmediato, involucrando a todo el equipo de desarrollo si fuera necesario.
- Verificar el funcionamiento correcto del *software* con pruebas automatizadas y ejecutadas frecuentemente.
- El 100% de las pruebas automatizadas deben ser satisfactorias para que los artefactos generados en la construcción puedan subirse al repositorio de control de versiones, si no se cumple esta práctica, se considera que el *software* construido es de mala calidad.
- Para evitar las construcciones rotas, el desarrollador debe obtener los cambios recientes del

equipo de desarrollo desde el repositorio de control de versiones y ejecutar localmente la integración del código.

Componentes de un sistema de Integración continua:

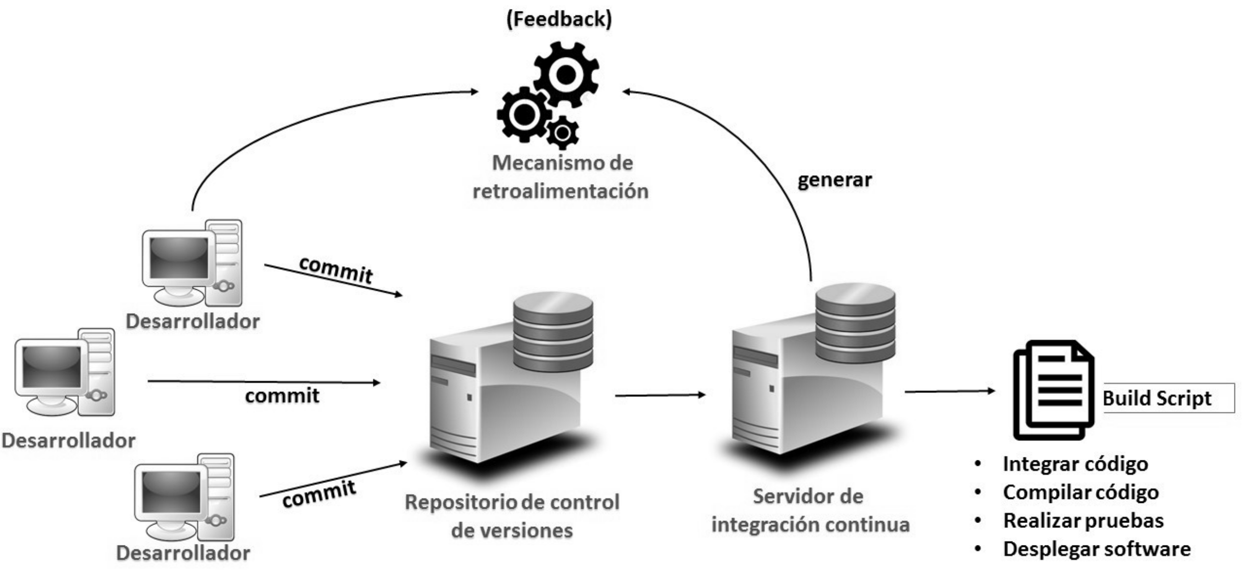



Figura 1: Componentes de un sistema de integración continua.

*(Fuente: Elaboración propia)*

- Desarrolladores: encargados de gestionar los paquetes en el repositorio, basándose en las prácticas anteriormente expuestas.
- Repositorio de control de versiones: permite al equipo de desarrollo trabajar en forma colaborativa, basándose en mantener todos los archivos del proyecto en un lugar centralizado, donde los desarrolladores se conectan y descargan una copia local del proyecto, enviando periódicamente los cambios que se realizan al servidor y se va actualizando su directorio de trabajo que otros usuarios a su vez han ido modificando.

- Servidor de Integración continua: es el orquestador de todo el proceso y debe estar configurado para comprobar los cambios en el repositorio de control de versiones cada minuto. Por cada construcción, el servidor genera reportes en el panel de control mostrando la salud del *software*.
- *Script* de construcción: conjunto de pasos para la construcción del *software* que son definidos a través de *scripts* que constan de una o varias secuencias para compilar, probar, inspeccionar y desplegar los artefactos en el repositorio.
- Mecanismos de retroalimentación: una vez finalizada la construcción, los resultados deben ser accesibles tan pronto como sea posible, brindando información respecto a la salud del *software* desarrollado a todo el equipo de desarrollo [9].

**Entrega continua y Despliegue continuo**

Existen prácticas asociadas al proceso de Integración continua como: la Entrega continua y el Despliegue continuo. El primero permite tener los artefactos generados en la Integración continua en estado listo para entregar a la fase de producción. Cuando la Entrega continua se implementa de manera adecuada, los desarrolladores dispondrán de un artefacto listo sometido anteriormente a un proceso de pruebas estandarizado [10]. Por otra parte el Despliegue continuo permite almacenar los artefactos generados en la Entrega continua en un repositorio de forma automática. La Figura 2 muestra la relación existente entre los procesos anteriormente mencionados.

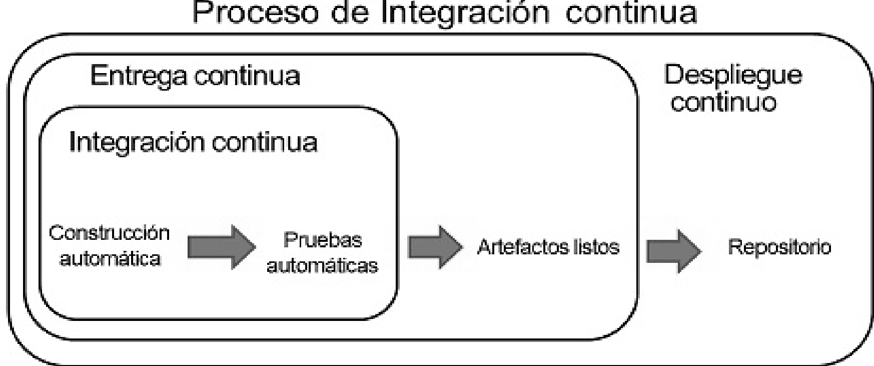



Figura 2: Proceso de Integración continua.

(Fuente: Elaboración propia)

## 1.4 Análisis de herramientas de Integración continua

El objetivo de usar herramientas de integración continua es que se encarguen de descargar el código de los repositorios, construir paquetes fuentes y binarios de forma automática, pasar las pruebas unitarias y por último guardar en un repositorio y mostrar los resultados.

**¿Cómo escoger la herramienta de Integración continua que más se adapte a la necesidad del centro?**

Para escoger la herramienta se define un plan de uso[4], que facilita determinar el estado actual de los sistemas informáticos a estudiar, permitiendo identificar la herramienta a utilizar en correspondencia con los objetivos a desarrollar mediante normas establecidas por el autor.

Plan de uso definido:

- Descartar las herramientas de Integración continua distribuidas bajo licencias privativas.

4 **Plan de uso:** conjunto de normas o reglas para guiar la selección de la herramienta.

- Estudiar herramientas de Integración continua con soporte para el controlador de versiones Git.
- Realizar un análisis investigativo para definir la mejor herramienta para la Integración continua de código abierto que se adapte a la necesidad del centro.

Según un estudio realizado en el 2016 sobre el porcentaje de uso de las diferentes empresas que han implantado la Integración continua, *Jenkins* es la más utilizada con un 60% a nivel mundial (ver Figura 3) [11].

Según el estudio realizado por la comunidad de Slant (*The Slant Community*[5]) la Tabla 1 muestra las mejores 10 herramientas de integración continua en la actualidad [12].

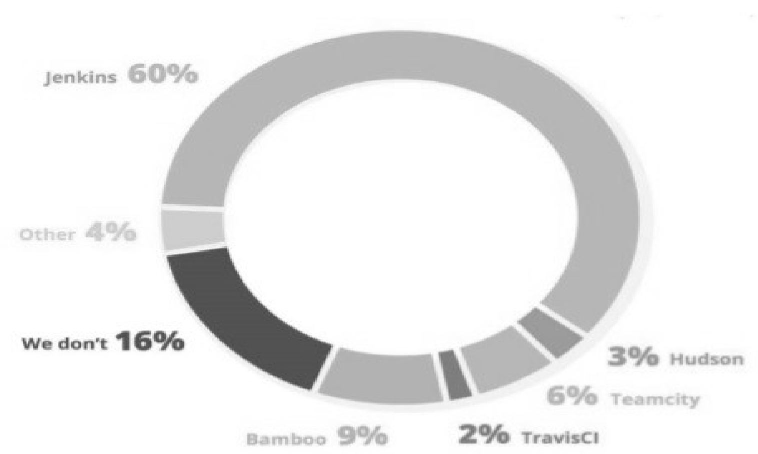



Figura 3: Porcentaje de uso de herramientas de integración continua a nivel mundial.

(Fuente: atlassian, 2016)

5 **Slant:** es una comunidad de recomendación de productos con el objetivo de que sea fácil encontrar el mejor producto para el cliente. Tiene más de 1.7 millones de seguidores que brindan información organizada a diario alrededor del problema que se desee resolver.

*Tabla 1: Las mejores 10 herramientas de integración continua según comunidades Slant.*

| No. | Herramientas | Soporte para Git | Licencia |
|---|---|---|---|
| 1 | Codeship | No | Privativa |
| 2 | CircleCI | No | Privativa |
| 3 | Travis | Si | Privativa |
| 4 | *Jenkins* | Si | Libre |
| 5 | Gitlab CI | Si | Libre |
| 6 | Wercker | No | Privativo |
| 7 | TeamCity | No | Privativa |
| 8 | SemaphoreCI | Si | Privativa |
| 9 | Bitrise | No | Privativa |
| 10 | Bamboo | No | Privativa |

Para el desarrollo de la solución se descartan Codeship y CircleCI debido a que no tienen soporte para Git y en el caso particular Travis CI debido a su limitación de uso específicamente con GitHub. Por lo cual se seleccionan *Jenkins* y GitLab CI como posibles herramientas a estudiar en cuanto a su condición de mejores herramientas de integración continua en la actualidad y por su soporte para Git. Además, la comunidad de Slant define en la Tabla 2 las mejores 5 herramientas de código abierto [12].

*Tabla 2: Las mejores 5 herramientas de integración continua para código abierto.*

| No. | Herramientas | Soporte para Git | Licencia |
|---|---|---|---|
| 3 | *Jenkins* | Si | Libre |
| 4 | Travis | Si | Libre |
| 5 | Concourse CI | No | Libre |

| | | | |
|---|---|---|---|
| 6 | GoCD | No | Libre |
| 7 | Buildbot | No | Libre |

En la tabla anterior se aprecia que *Jenkins* ocupa el 1er lugar entre las mejores cinco herramientas de Integración continua para código abierto, se descartan las restantes ya que no tienen soporte para Git como controlador de versiones. Una vez analizada las condiciones que cumplen cada herramienta y rigiendo la investigación en cuanto al plan de uso definido anteriormente, se decide: estudiar y comparar Git Lab CI y *Jenkins*.

*Tabla 3: Comparación GitLab CI y Jenkins.*

| **Aspectos** | **GitLab CI** | **Jenkins** |
|---|---|---|
| Configuración por rama | - Basado en un modelo de fichero de configuración escrito en YAML (tipo de archivo: travis.yml) y almacenado en la raíz de cada repositorio.<br>- Fomenta que cada rama tenga su propia configuración y permite a los colaboradores aportar configuraciones.<br>- Permite probar nuevos flujos de trabajo en sus ramas personales o de funciones, sin correr el riesgo de afectar las interconexiones existentes y las que están en funcionamiento, que se usan para las ramas generales como *master*.<br>- El trabajo se envía a máquinas llamadas corredores, que son fáciles de configurar y se pueden aprovisionar en muchos sistemas operativos diferentes. | - Proceso de Integración continua que se puede definir de manera declarativa o imperativa utilizando el lenguaje Groovy (tipo de archivo: *Jenkinsfile*) dentro del propio repositorio o mediante cuadros de texto en la interfaz web de *Jenkins*.<br>- El *Multibranch pipeline* es un *plugin* que se puede definir en el propio Git del proyecto y mantenerse en el propio sistema de control de versiones el historial de cambios del *pipeline* (permite crear múltiples ramas).<br>- Permite la construcción del *pipeline* en la interfaz web del usuario.<br>- Permite configurar al detalle los proyectos (por ejemplo, crear trabajos que hagan uso de más de un repositorio en GitLab). |

| | | |
|---|---|---|
| Tablero(*dashboard*) | - Solo muestra la última compilación en la lista de proyectos y en proyectos individuales muestra líneas completas con los *log* de todas sus etapas.<br>- No muestra los trabajos específicos dentro de las etapas, por lo que puede ser un poco inexpresivo. | - Muestra información sobre la cantidad de compilaciones que tuvieron éxito o fallo, y permite monitorear el flujo del proceso en tiempo real.<br>- Brinda información de cada etapa de la interconexión de forma detallada, si tuvieron éxito o no, cada una de las etapas definidas previamente en el *pipeline* y en caso de error muestra una posible solución. |
| Generación de informes | - Es limitado, ya que solo es posible extraer una parte específica mediante expresiones regulares del resultado de la compilación y la prueba de la consola. | - Genera informes con los *log* de las pruebas y los detalles de cada una de las etapas por la que atraviesa el *software.* |
| Historia Visual de Informes | - Muestra un pequeño gráfico del historial del informe de prueba directamente en la página de detalles del proyecto.<br>- Permite ver lo que se está implementando actualmente en sus servidores y acceder a una vista detallada de todas las implementaciones anteriores.<br>- Desde la lista se puede volver a implementar la versión actual, o incluso deshacer una antigua en caso de que algo salga mal. | - Muestra en la página de interconexión cada paso que se ejecuta en cada una de sus etapas de forma detallada en tiempo real.<br>- Permite navegar por la historia de *builds* del proyecto.<br>- Tiene la capacidad de mostrar el informe de compilación o prueba directamente en su página de interconexión. |

| Soporte de *Plugins* | - Actualmente no cuenta con soporte de *plugins*. | - Existen más de 800 *plugins* para *Jenkins*.<br>- Debido a su gran flexibilidad y compatibilidad con numerosos *plugins*, permiten crear flujos completos capaces de implementar sistemas de Entrega continua y Despliegue continuo. |
|---|---|---|

Como resultado de la tabla comparativa anterior se determinó que GitLab CI es un componente fijo del gestor de repositorios Git y por tanto, ofrece interacción entre los procesos de Integración continua y la funcionalidad del repositorio. *Jenkins*, por el otro lado, está ligeramente acoplado de cualquier administrador de repositorios y es muy flexible cuando se trata de la selección de sistemas de control de versiones. Además, *Jenkins* hace hincapié en el soporte de complementos para ampliar o mejorar la funcionalidad existente del *software*. Por lo mencionado anteriormente y por la experiencia del usuario en cuanto a su uso, se decide utilizar como herramienta de Integración continua: ***Jenkins***.

### 1.4.1 Herramienta de Integración continua (*Jenkins*)

*Jenkins* permite planificar y realizar multitud de tareas, simplificando los procesos involucrados en el ciclo de vida de un proyecto. Es distribuido bajo licencia MIT[6] y permite llevar a cabo las siguientes acciones:

- Construcción continua y pruebas automatizadas de proyectos *software*.
- Monitorización de la ejecución de servicios externos.
- Despliegue automático.

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. Permite crear y configurar los llamados '*jobs*', que es el conjunto de tareas parametrizables que definirán el proceso de Integración continua a realizar en

6 **MIT:** licencia de software que se origina en el Instituto de Tecnológico de Massachusetts.

un proyecto concreto [13]. Además puede trabajar con el *pipeline y c*uando sea necesario, se ejecutarán las tareas descritas en dicho fichero. Las tareas pueden ser ejecutadas en el mismo servidor de *Jenkins* o pueden ser desplegadas en máquinas esclavas y puede realizar trabajos en paralelo en una misma máquina.

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, compilar el código y ejecutar las pruebas para testearlo.
- Notificación de errores que se han ejecutado tras la ejecución de pruebas, por ejemplo, vía *mail, twitter, chat*.
- Multiplataforma.
- Permite variar la manera de notificar errores.
- Integración con bases de datos.
- Personalizar interfaz.
- Permite la creación de extensiones a través de Java.
- Huellas digitales de archivos: se realiza seguimiento a los archivos del proyecto.
- Permite compilaciones y pruebas distribuidas.
- Comunidad de soporte.
- Integración de correo electrónico, generando reportes o notificaciones.

*Jenkins* brinda una serie de ventajas en la instalación y configuración de la herramienta, que permite gestionar el proceso a través del administrador del servidor desde su interfaz gráfica, desde la cual se puede controlar los siguientes elementos:

- Actualizar configuración desde la configuración desde el disco duro.
- Administrar *plugins*.
- Muestra la información del sistema.
- Registro del sistema a través de *logs* que captura todas las salidas.
- Se puede administrar o acceder a *Jenkins* desde la consola o desde *scripts.*

*Jenkins* constituye uno de las herramientas de Integración continua más completas, se destaca debido a su alta funcionalidad e integración con muchos entornos, una importante extensibilidad llevada a cabo mediante *plugins* y por la cantidad de documentación que existe de esta herramienta [13].

***Pipeline***

Una forma de implantar la Integración continua es mediante una abstracción denominada *pipeline*. Un *pipeline* trata de modelar el conjunto de fases y tareas automatizadas que se deben llevar a cabo en el proceso que va desde la obtención del *software* hasta que se despliega en producción. Dichas tareas y fases se definen en forma de código, ejecutándose de manera secuencial, siendo la entrada de cada una la salida de la anterior, de manera que si se produce un fallo en cualquier etapa deja de ejecutarse los pasos posteriores [14]. En el desarrollo de la solución, las fases por las que transcurre el *pipeline* viene representado por la siguiente figura:

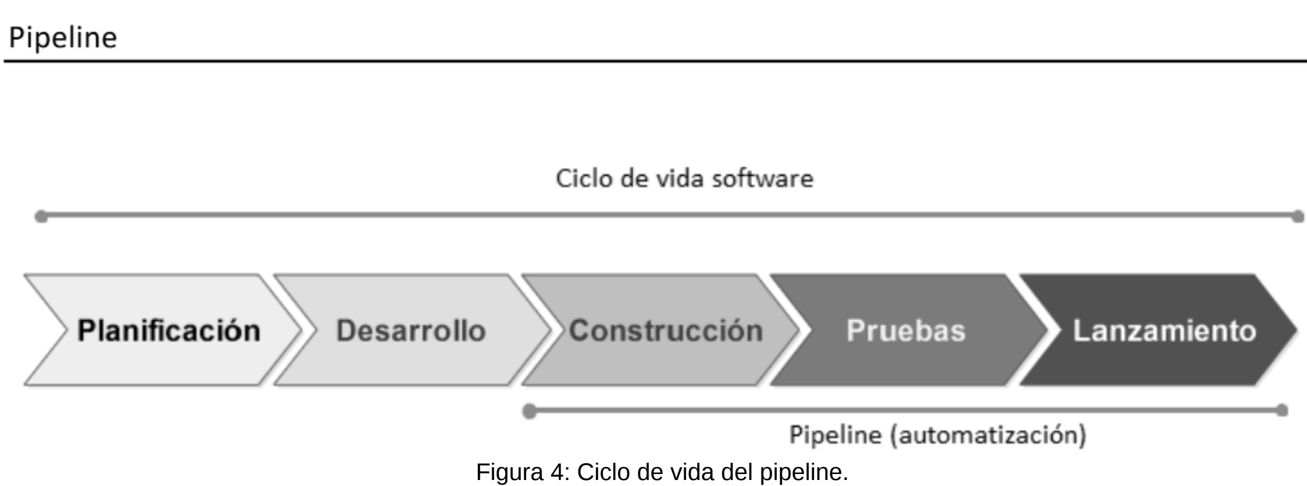



Figura 4: Ciclo de vida del pipeline.

*(Fuente: Bersabé, 2016)*

## 1.5 Metodología para guiar el desarrollo de la solución

Una metodología de desarrollo de *software* es un conjunto integrado de técnicas y métodos que permite abordar de forma homogénea y abierta cada una de las actividades del ciclo de vida de un proyecto de desarrollo. Es un proceso de software detallado y completo. Las metodologías se basan en una combinación de los modelos de proceso genéricos. Definen artefactos, roles y actividades, junto con prácticas y técnicas recomendadas [15].

Para el desarrollo de la solución se utiliza como metodología AUP variación para la UCI, adaptada a las necesidades de los centros productivos de la Universidad de las Ciencias Informáticas para lograr una mejor estructura y planificación en el desarrollo del *software*.

### 1.5.1 AUP variación para la UCI

Es una metodología ágil, basada en AUP, a través de su utilización se promueve el trabajo en equipo, se basa en la retro alimentación continua entre el cliente y equipo de desarrollo y la comunicación fluida entre todos los participantes [16]. Está compuesta por tres fases:

- Inicio: durante el inicio del proyecto se llevan a cabo las actividades relacionadas con la planeación del proyecto. En esta fase se realiza un estudio inicial de la organización cliente que permite obtener información fundamental acerca del alcance del proyecto, realizar estimaciones de tiempo, esfuerzo y costo y decidir si se ejecuta o no el proyecto.
- Ejecución: 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 le realizan las pruebas y por último se despliega 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 [16].

A partir de que el Modelado de negocio propone tres variantes a utilizar en los proyectos (Casos de Uso del Negocio, Descripción de Proceso de Negocio o Modelo Conceptual) y existen tres formas de encapsular los requisitos (Casos de Uso del Sistema, Historias de usuario, Descripción de requisitos por proceso), surgen cuatro escenarios para modelar el sistema en los proyectos. Para modelar la propuesta de solución, se utiliza el escenario 4 que trabaja con Historias de Usuario (HU) dadas las similitudes en sus características con el *software* a desarrollar. Dicho escenario se recomienda en proyectos no muy extensos, ya que una HU no debe poseer demasiada información.

## 1.6 Lenguajes de programación y herramientas a utilizar

Para llevar a cabo la solución es necesario organizar el ambiente de desarrollo, definiendo los lenguajes de programación a utilizar y un conjunto de herramientas.

### 1.6.1 Lenguajes de programación

Los lenguajes de programación se utilizan para controlar el comportamiento de una máquina, particularmente una computadora. Consisten en un conjunto de reglas sintácticas y semánticas que definen su estructura y el significado de sus elementos, respectivamente. En la actualidad existe una gran

variedad de lenguajes de programación utilizados en el desarrollo de *software* libre.

**Bash (*Bourne again shell*):** es un programa informático, cuya función consiste en interpretar órdenes en un lenguaje de 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 que ejecuta comandos desde la entrada estándar o de un archivo, también incorpora características útiles de los shells Korn y C (ksh y csh) [17]. En el desarrollo de la solución se implementarán *scripts* de bash que guiarán el proceso de construcción y publicación de paquetes fuentes y binarios, donde cada *script* se corresponderá con una etapa del proceso.

**Groovy:** lenguaje de programación dinámico orientado a objetos para la máquina virtual Java (JVM). Tiene una sintaxis similar al lenguaje Java, una curva de aprendizaje corta, soporte para lenguajes específicos de dominio (*domain-specific languages*), sintaxis compacta, potentes primitivas de procesamiento, la facilidad de desarrollo de aplicaciones *Web*, soporte para las pruebas unitarias entre otras. En el desarrollo de la solución, el archivo *jenkinsfile*[7] se implementará con la sintaxis de Groovy, donde se definirá las etapas del proceso con este lenguaje[18].

### 1.6.2 Sistema de control de versiones Git

Git es un sistema de control de versiones de código abierto y gratuito diseñado para manejar pequeños y grandes proyectos, con velocidad y eficiencia. Supera a las herramientas de SCV como Subversion, CVS, Perforce y ClearCase con funciones como ramificación local barata, áreas de preparación conveniente y flujos de trabajo múltiples. En cuanto a derechos de autor Git es un *software* libre distribuible bajo los términos de la versión 2 de la Licencia Pública General de GNU. La característica de Git que realmente lo distingue de casi todos los demás sistemas que existen es su modelo de ramificación. Permite tener múltiples ramas locales que pueden ser totalmente independientes entre sí, donde la creación, fusión y

---

7 **Jenkinsfile:** archivo pipeline donde se define el flujo de trabajo.

eliminación de esas líneas de desarrollo lleva segundos. Todo en Git es verificado mediante una suma de comprobación (*checksum*) antes de ser almacenado, y es identificado a partir de ese momento mediante dicha suma. Esto significa que es imposible cambiar los contenidos de cualquier archivo o directorio sin que Git lo sepa. Esta funcionalidad está integrada al más bajo nivel en su interior y es parte integral de su filosofía. No se puede perder información durante su transmisión o sufrir corrupción de archivos sin que Git lo detecte [6].

Git tiene tres estados principales en los que se pueden encontrar los archivos: confirmado (*committed*), modificado (*modified*) y preparado (*staged*). Confirmado significa que los datos están almacenados de manera segura en la base de datos local. Modificado que se ha modificado el archivo, pero todavía no se ha confirmado a la base de datos local. Y preparado que se ha marcado un archivo modificado en su versión actual para que vaya en la próxima confirmación. Un proyecto de Git cuenta con cuatro secciones principales: el directorio de trabajo, el repositorio local, el repositorio remoto y el área de preparación (*staging area*) [19] lo cual se puede apreciar en la Figura 5.

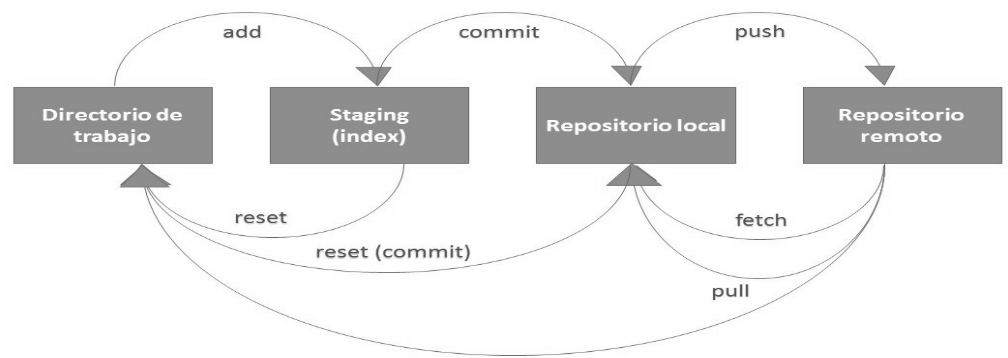



Figura 5: Flujo de trabajo básico en Git.

*(Fuente: GIT, 2014)*

- Directorio de trabajo: es una copia de una versión del proyecto. Estos archivos se sacan de la base de datos comprimida en el directorio de Git y se coloca en disco de la máquina para que se pueda

usar o modificar.

- Área de preparación (*staging area*): es un archivo, generalmente contenido en el directorio de Git, que almacena información acerca de lo que va a ir en la próxima confirmación. En algunas ocasiones o algunos autores a ésta la denominan “índice”, pero cada vez es más frecuente referirse a ella como “área de preparación”.
- Repositorio remoto: es la parte más importante de Git, lugar donde se almacenan los metadatos y la base de datos de objetos para el proyecto a desarrollar, al cual el desarrollador accede para descargar el paquete o una de sus versiones a gestionar.
- Repositorio local: almacenan los metadatos y la base de datos de objetos para el proyecto de forma local.

**Estándar DEP 14**

Formato recomendado para armonizar el diseño de los repositorios de Git utilizados para mantener los paquetes de Debian.  Persigue múltiples objetivos tales como: facilitar para Debian y sus derivados la construcción de paquetes sobre sus respectivos repositorios Git y permitir el cambio entre varias herramientas de ayuda de empaquetado incluso si todas las herramientas no implementan el mismo flujo de trabajo [20]. DEP 14 establece que el trabajo en el repositorio Git estará estructurado en forma de árbol con tres ramas principales:

- ***Master*****:** es la rama que se crea por defecto al crear un nuevo proyecto en Git. Mantiene las versiones oficiales de los proyectos, almacena los *release* y los *tag* correspondiente.
- ***upstream*****:** rama donde se guarda el código fuente original del paquete de un desarrollador. Genera etiquetas de tipo *upstream* / <*version*> creada por el mantenedor del paquete cuando sea necesario: por ejemplo: cuando hace una versión basada en una instantánea de Git.
- ***pristine-tar:*** puede regenerar una copia exacta de un *tarball* original *upstream* y el contenido del *tarball* utilizando solo un pequeño archivo delta binario, que normalmente se mantienen en una

rama *upstream* en el control de versión. El archivo delta está diseñado para ser controlado en el control de versiones a lo largo de la rama *upstream*, lo que permite que los paquetes de Debian se construyan completamente utilizando fuentes del SCV, sin la necesidad de mantener copias de los archivos secundarios. La rama *pristiene-tar* soporta un *tarball* comprimido, llamando a prístino-gz, prístino-bz2 y prístino-xz para producir los archivos prístinos gzip, bzip2 y xz, hallando la diferencia entre un comprimido y otro para generar un *upstream* exactamente igual.

### 1.6.3 Dpkg

La herramienta Dpkg es la base del sistema de gestión de paquetes de Debian GNU/Linux. Fue creado por Ian Jackson en 1993. Se utiliza para instalar, quitar y proporcionar información sobre los paquetes .deb. Es una herramienta de bajo nivel; se necesita un frontal de alto nivel para traer los paquetes desde lugares remotos o resolver conflictos complejos en las dependencias de paquetes. Distribuida bajo la licencia: GNU General Public License [5].

### 1.6.4 Sbuild

Herramienta encargada de reconstruir los paquetes binarios de Debian instalando las dependencias de origen que falten. La construcción se realiza en un entorno de compilación limpio dedicado (*chroot*). Sbuild puede buscar el origen Debian en una red, o puede usar fuentes disponibles localmente. Recibe un paquete para procesar de tipo paquete.dsc con un nombre de paquete fuente junto con una versión en el formato paquete_versión. También es posible ejecutar comandos externos y puede construir usando un paquete local con su archivo (.dsc) o uno remoto especificando una versión de dpkg explícita [5].

### 1.6.5 Gbp

Integra el sistema de compilación del paquete Debian con el controlador de versiones Git. Es una herramienta que permite el importe de un paquete Debian existente en un repositorio de ese tipo, permite incrementar nuevas versiones de un paquete y añadir modificaciones posteriores. Con gbp las ramas se pueden cambiar en cualquier momento y puedes trabajar con un número arbitrario. Como el paquete de

construcción de gbp solo funciona con repositorios locales de Git, se debe usar *git push* para publicar los cambios en un repositorio remoto [5].

### 1.6.6 Visual Paradigm

Visual Paradigm para UML 8.0 es una herramienta profesional que soporta el ciclo de vida completo del desarrollo de *software*. Es multiplataforma, utiliza UML como lenguaje de modelado y cuenta con una versión libre para la comunidad, ayudando de una manera rápida a la construcción de aplicaciones de mayor calidad y a un menor costo. Esta herramienta proporciona una interfaz gráfica de usuario y una base de datos de esquema de apoyo [21]. Se decide utilizar para modelar los diagramas que se generen en cada una de las etapas del proceso de desarrollo de *software* ya que cuenta con una amplia gama de funcionalidades y con facilidad de uso, además de ser la que está definida por la universidad y por la que la misma paga una licencia con fines educacionales.

### 1.6.7 Aptly

Es una herramienta para crear repositorios remotos los cuales pueden ser parciales o completos, permite administrar repositorios de forma local, filtrarlos, combinarlos, actualizar paquetes individuales, tomar instantáneas y publicarlos de nuevo como repositorios de Debian. El objetivo de *aptly* es almacenar las versiones de los paquetes y controlar los cambios en un entorno centralizado. Al mismo tiempo realiza un control y mantiene los cambios detallados en los contenidos del repositorio para hacer la transición del entorno de su paquete a una nueva versión utilizando los *mirrors* [5].

### 1.6.8 Piuparts

Herramienta para probar que los paquetes .deb puedan ser instalados, actualizados y eliminados sin problemas. Lo hace creando un Debian con una mínima instalación, actualización y eliminación de paquetes en ese entorno, al terminar compara el estado del directorio con un árbol de un antes y un después. Piuparts informa sobre cualquier archivo que se haya agregado, eliminado o modificado durante este proceso. Se entiende como una herramienta de control de calidad para las personas que crean

paquetes .deb que permite probarlos antes de subirlos al repositorio. Está disponible bajo la Licencia Pública General de GNU, versión 2, o cualquier versión posterior [5].

### 1.6.9 Lintian

Herramienta que verifica que no existan errores en los paquetes de Debian, ni violaciones de políticas y en caso de existir genera un informe. Utiliza un directorio de archivos, llamado laboratorio, en el que almacena información sobre los paquetes que examina. Puede mantener la información de múltiples invocaciones para evitar repetir costosas operaciones de recolección de datos [5].

## 1.7 Conclusiones parciales

El estudio del proceso de construcción y publicación de paquetes fuentes y binarios para la Distribución Cubana GNU/Linux Nova, permitió determinar que es ineficiente, carece de seguridad y que no existe una herramienta definida para agilizar dicho proceso. A través del estudio de las herramientas de Integración continua se determinó utilizar *Jenkins* para el desarrollo de la solución. Se define DEP 14 como estándar de ramificación para el trabajo con la herramienta de control de versiones Git.

# Capítulo 2: Concepción de la propuesta de solución

## 2.1 Introducción

En el capítulo se representa el modelo de dominio para obtener la relación entre los conceptos en el desarrollo de la implementación. Además, se hace un análisis sobre las características y el diseño que tiene la propuesta de solución. Se definen las técnicas de obtención de requisitos, permitiendo precisar los requisitos funcionales y no funcionales. Se realiza la validación de los requisitos y se describen de forma detallada las especificaciones de los requisitos en las historias de usuarios.

## 2.2 Modelo de dominio

El Modelo de Dominio o Modelo Conceptual, permite de manera visual mostrar al usuario los principales conceptos que se manejan en el dominio del problema. Representa las clases conceptuales del mundo real, no de componentes de *software*. Puede considerarse como un diccionario visual de las abstracciones relevantes, vocabulario e información del dominio, aprovechando las oportunidades de los diagramas UML para representar conceptos [22].

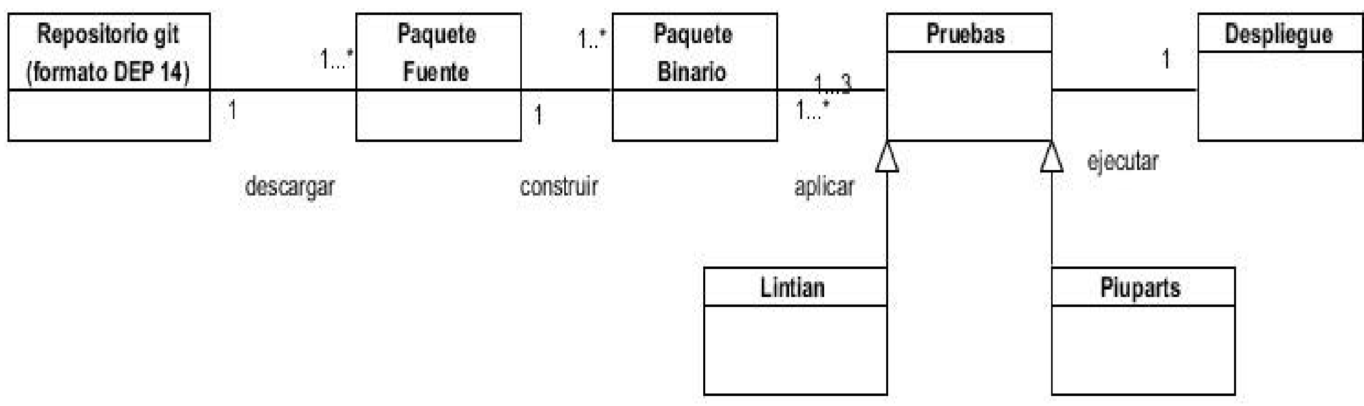



Figura 6: Modelo de dominio

*(Fuente: Elaboración propia)*

**Repositorio Git:** almacena un registro de los cambios en los paquetes y coordina el trabajo para el equipo de desarrollo sobre los archivos compartidos.

**Paquete fuente:** paquete que se construye a partir de los cambios realizados en el proyecto Git.

**Paquete binario:** paquete que se construye a partir del paquete fuente

**Pruebas Lintian:** utilizada para comprobar que no existan errores ni violaciones de políticas en los paquetes construidos.

**Pruebas Piuparts:** comprueba que los paquetes .deb puedan ser instalados, actualizados y eliminados sin problemas. Informa sobre cualquier archivo que se haya agregado, eliminado o modificado durante el proceso.

**Despliegue:** proceso mediante el cual se almacenan los artefactos en el repositorio.

## 2.3 Condiciones para el funcionamiento de la solución

La solución a desarrollar debe tener acceso al repositorio para obtener las dependencias utilizadas en el proceso de construcción del paquete fuente. Poseer las herramientas GBP, Sbuild, Piuparts, Lintian, Aptly, *Jenkins* y el repositorio Git estandarizado por el formato DEP 14. El archivo *Jenkins*file debe estar incluido en la carpeta Debian del código fuente del paquete. Los scripts *buildsource*, *buildpackage*, *test_lintian*, *test_piuparts*, *archive_actefactos* y *publish* deben almacenarse en */usr/share/integracion-continua.*

## 2.4 Requisitos del sistema

Los requisitos del sistema describen la función y las características de un sistema y las restricciones que guían su desarrollo [23]. Para el desarrollo de la propuesta de solución se identificaron requisitos funcionales y no funcionales para establecer una comunicación más específica entre el cliente y el desarrollador.

### 2.4.1 Definición de las técnicas de obtención de requisitos

Existen varias técnicas que permiten establecer comunicación clara y eficiente entre los interesados en el producto y el equipo de trabajo. Estas técnicas se centran principalmente en el descubrimiento de los requisitos del sistema. En la identificación de los requisitos de la propuesta de solución se utilizaron las siguientes técnicas:

**Entrevista:** utilizada con el fin de obtener diversas opiniones y recomendaciones en un intercambio mediante preguntas con los especialistas de CESOL, para esclarecer con precisión el funcionamiento del proceso de construcción y publicación de paquetes.

**Tormenta de ideas:** se utilizó como herramienta de trabajo grupal facilitando la obtención de los requisitos funcionales y no funcionales para desarrollar la implementación el proceso de construcción y publicación de paquetes fuentes y binarios para la Distribución Cubana GNU/Linux Nova.

### 2.4.2 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 [23].

- **RF 1:** Empaquetar código fuente.
- **RF 2:** Construir paquete binario para las diferentes arquitecturas.
- **RF 3***:* Realizar la prueba de piuparts para el paquete binario.
- **RF 4:** Realizar la prueba de lintian para el paquete binario.
- **RF 5:** Archivar los artefactos generados en la construcción.
- **RF 6:** Almacenar los artefactos generados en el repositorio.

### 2.4.3 Requisitos no funcionales.

Según Somerville los requisitos no funcionales o requerimientos no funcionales son las limitaciones sobre servicios o funciones que ofrece el sistema. Los requerimientos no funcionales se suelen aplicar al

sistema como un todo, más que a características o a servicios individuales del sistema [24].

**Requisito de *hardware*:**

- **RNF 1:** el servidor debe contar como mínimo con una capacidad de 1TB de almacenamiento.
- **RNF 2:** el servidor debe contar como mínimo con un procesador Intel Core i3-4150 a 3.0GHz y 8 GB de RAM.

**Requisito de software:**

- **RNF 3:** el servidor debe tener instalado las herramientas *Jenkins*, GBP, Sbuild, Piuparts, Lintian, Aptly, *Jenkins.*
- **RNF 4:** el repositorio Git debe estar estandarizado por el formato DEP 14.

**Requisito de Seguridad:**

- **RNF 5:** la herramienta debe contar con un control de acceso basado en roles a través de un usuario y una contraseña. El rol administrador de la configuración es el único que administra todo el flujo de trabajo en la herramienta y posee la llave primaria para la firma de los paquetes.

### 2.4.4 Validación de los requisitos

La validación de requisitos examina las especificaciones para asegurar que todos los requisitos del sistema han sido establecidos sin ambigüedad, sin inconsistencias, sin omisiones, que los errores detectados hayan sido corregidos, y que el resultado del trabajo se ajusta a los estándares establecidos para el proceso, el proyecto y el producto [23]. Para la validación de los requisitos identificados en la investigación se utiliza la técnica de Casos de prueba. Los parámetros a validar en los requisitos son [24]:

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

- **Completitud:** se deben incluir todos los requisitos que definan todas las funciones y restricciones propuestas por el usuario del sistema.
- **Realismo:** los requisitos deben verificarse que se pueden implementar con la tecnología existente.
- **Verificabilidad:** los requisitos deben redactarse de forma que puedan ser verificables para evitar discusiones entre el cliente y el equipo de desarrollo.

## 2.5 Historias de usuarios

Las historia de usuario son la técnica utilizada para especificar los requisitos del *software*. Se trata de tarjetas de papel en las cuales el cliente describe brevemente las características que el sistema debe poseer, sean requisitos funcionales o no funcionales. Cada historia de usuario es lo suficientemente comprensible y delimitada para que los programadores puedan implementarlas en unas semanas. Las historias de usuario se descomponen en tareas de programación y se asignan a los programadores para ser implementadas durante una iteración [25].

Las historias de usuarios conforman la parte central del escenario 4 de la metodología AUP variación para la UCI, pues definen lo que se debe construir en el proyecto de *software*. Tienen una prioridad (Alta, Media, Baja) asociada, definida por el cliente y la importancia que tiene para el funcionamiento de la herramienta las funcionalidades antes mencionadas. El tiempo de cada funcionalidad será estimado por el desarrollador. La estimación del tiempo está dada de acuerdo al tiempo que toma la tarea en ser desarrollada. A continuación, se describen las historias de usuario para el desarrollo de la herramienta [16].

*Tabla 4: Historia de usuario #1.*

| **Historia de Usuario** | |
|---|---|
| **Número:** RF 1 | **Usuario:** Administrador |
| **Nombre de historia:** Empaquetar código fuente. | |

| **Prioridad:** Alta | **Riesgo en desarrollo:**<br>- Que no cumpla con el formato DEP 14. |
|---|---|
| **Tiempo estimado:** 4 semanas | |
| **Programador responsable**: Manuel Alejandro Ricardo Serrano. | |
| **Descripción:** haciendo uso de la herramienta GBP empaquetar el código fuente. | |
| **Observaciones:** Para que funcione, el repositorio debe tener el formato DEP 14. Lo que se va obtener es un paquete fuente modificado por el desarrollador. | |

*Tabla 5: Historia de usuario #2.*

| **Historia de Usuario** | |
|---|---|
| **Número:** RF 2 | **Usuario:** Administrador |
| **Nombre de historia:** Construir paquete binario para las diferentes arquitecturas. | |
| **Prioridad:** Media | **Riesgo en desarrollo:**<br>- Existan errores de dependencias de construcción.<br>- Pérdida de energía eléctrica durante el proceso de construcción del paquete binario.<br>- Fallas durante la construcción porque el paquete no se puede ejecutar dentro del *chroot*. |
| **Tiempo estimado:** 3 semanas. | |
| **Programador responsable**: Manuel Alejandro Ricardo Serrano. | |
| **Descripción:** Haciendo uso de la herramienta de construcción de paquetes *sbuild* construir a partir de un paquete fuente, el paquete binario para la arquitectura i386 y amd64. | |
| **Observaciones:** Si no hay una base de *sbuild*, hay que crearla. | |

*Tabla 6: Historia de usuario #3.*

| **Historia de Usuario** | |
|---|---|
| **Número:** RF 3 | **Usuario:** Administrador |
| **Nombre de historia:** Realizar la prueba de Lintian para el paquete binario. | |
| **Prioridad:** Media | **Riesgo en desarrollo:**<br>- Herramientas desactualizadas o que no estén disponible en el repositorio. |
| **Tiempo estimado:** 3 semanas. | |
| **Programador responsable**: Manuel Alejandro Ricardo Serrano. | |
| **Descripción:** la herramienta de prueba Lintian verifica que no existan errores y violaciones de políticas en los paquetes. | |
| **Observaciones:** prueba aplicada para verificar la construcción de los paquetes binarios. | |

*Tabla 7: Historia de usuario # 5.*

| **Historia de Usuario** | |
|---|---|
| **Número:** RF 5 | **Usuario:** Administrador |
| **Nombre de historia:** Realizar la prueba de Piuparts para el paquete binario. | |
| **Prioridad:** Media | **Riesgo en desarrollo:**<br>**-** Herramientas desactualizadas o que no esté disponible en el repositorio. |
| **Tiempo estimado:** 3 semanas. | |
| **Programador responsable**: Manuel Alejandro Ricardo Serrano. | |
| **Descripción:** la herramienta de prueba que los paquetes pueden ser instalados, actualizados y eliminados sin | |

| problema. |
|---|
| **Observaciones:** Prueba aplicada para verificar que los paquetes se instalen, se actualicen y se eliminen. |

*Tabla 8: Historia de usuario # 6.*

| **Historia de Usuario** | |
|---|---|
| **Número:** RF 6 | **Usuario:** Administrador |
| **Nombre de historia:** Archivar los artefactos generados en la construcción. | |
| **Prioridad:** Alta | **Riesgo en desarrollo:**<br>- Falla en el proceso de construcción de artefactos en etapas anteriores, en consecuencia se archivan artefactos defectuosos. |
| **Tiempo estimado:** 4 semanas. | |
| **Programador responsable**: Manuel Alejandro Ricardo Serrano. | |
| **Descripción:** se definen en el archivo *Jenkinsfile* los artefactos a archivar. | |
| **Observaciones:** se archivan los .deb, .lintian, .piuparts, .gz, .xz, .bz2, .lzma, .dsc, .build, .changes. | |

*Tabla 9: Historia de usuario # 7.*

| **Historia de Usuario** | |
|---|---|
| **Número:** RF 7 | **Usuario:** Administrador |
| **Nombre de historia:** Almacenar los artefactos generados en el repositorio. | |
| **Prioridad:** Alta | **Riesgo en desarrollo:**<br>- Desactualización de la herramienta o que no esté disponible en el repositorio. |

| **Tiempo estimado:** 4 semanas. |
| --- |
| **Programador responsable**: Manuel Alejandro Ricardo Serrano. |
| **Descripción:** se publican en los repositorios los *snapshots* y los *release*. |
| **Observaciones:** para el trabajo en la publicación de repositorios se utiliza la herramienta Aptly. |

## 2.6 Definición de la arquitectura

De acuerdo al *software* Engineering Institute (SEI), la Arquitectura de *software* se refiere a “las estructuras de un sistema, compuestas de elementos con propiedades visibles de forma externa y las relaciones que existen entre ellos [26]. Para el desarrollo de la solución se define como arquitectura a utilizar Flujo de datos, la cual se aplica cuando los datos de entrada son transformados a través de una serie de componentes en datos de salida [23].

### 2.6.1 Patrón arquitectónico

Los datos de entrada son transformados a través de una serie de componentes computacionales en los datos de salida. Un patrón tubería y filtro tiene un grupo de componentes llamados filtros, conectados por tuberías que transmiten datos de un componente al siguiente. El filtro está diseñado para recibir entrada de datos de una forma y producir la salida de datos de una forma específica [23].

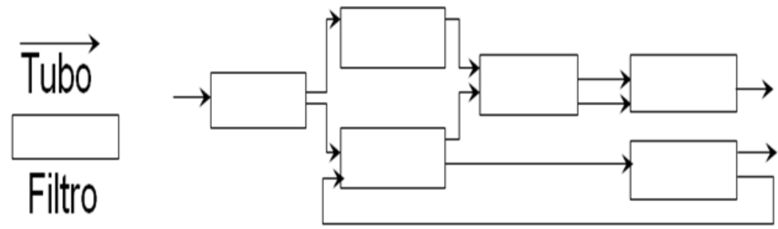



Figura 7: Patrón arquitectónico Tuberías y filtros.

(Fuente: Elaboración propia)

## 2.7 Descripción de la propuesta de solución

La propuesta de solución tiene entre sus principales objetivos garantizar una correcta implementación de práctica de Integración continua, haciendo uso de la herramienta *Jenkins* para el proceso de construcción y publicación de paquetes fuentes y binarios en la Distribución Cubana GNU/Linux Nova.

Para que la Integración continua con la herramienta *Jenkins* pueda iniciar su ejecución, el desarrollador debe cumplir con la premisa de incluir en la carpeta *debian* del paquete a gestionar, el archivo *jenkinsfile*. Archivo que tendrá las tareas programadas que permiten realizar acciones de forma automática para ejecutar la Integración continua. El código que poseé dicho archivo es utilizando un lenguaje basado en Groovy, está estructurado por seis etapas *(buildsource*, *buildpackage*, *test_lintian*, *test_piuparts*, *archive_actefactos* y *publish*).

En la etapa de *buildsource* se construye el paquete fuente con la herramienta gbp, en *buildpackage* a partir del paquete fuente obtenido en la primera etapa se construyen dos paquetes binarios, uno para la arquitectura i386 y otro para amd64. Luego los paquetes binarios serán sometidos a dos tipos de pruebas, lintian y piuparts, tercera y cuarta etapa respectivamente. En la quinta etapa, se le ordena a Jenkins generar los artefactos, por último, se publican con la herramienta aptly los *snapshots* y *release* en los repositorios correspondientes. La Error: no se encontró el origen de la referencia muestra el flujo del proceso de Integración continua en la herramienta Jenkins.

- La interfaz de la herramienta *Jenkins* adquiere un color verde cuando el proceso se ejecuta correctamente.
- La interfaz de la herramienta *Jenkins* adquiere un color rojo cuando el proceso no se ejecuta de manera satisfactoria y muestra una X en rojo en la etapa fallida.
- La interfaz de la herramienta *Jenkins* adquiere un color gris cuando se interrumpe el proceso mostrando también una X en rojo en la etapa detenida.

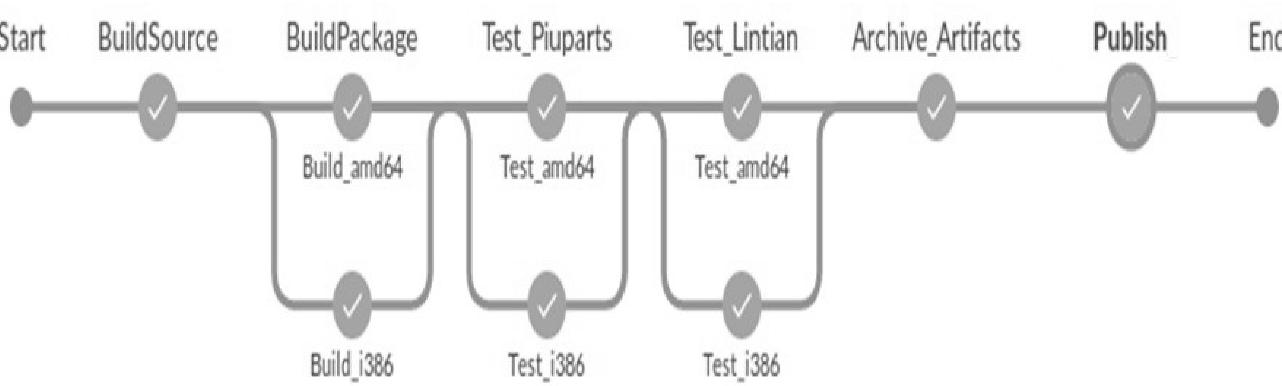



Figura 8: Flujo del proceso de Integración continua.

(Fuente: elaboración propia)

## 2.8 Conclusiones parciales

Con la elaboración del modelo conceptual se logró mostrar visualmente los principales conceptos, obteniendo un mejor dominio y compresión del problema para identificar las posibles soluciones. Se identificaron mediante las técnicas de obtención de requisitos seis funcionales y cinco no funcionales, los cuales constituyeron una guía fundamental para la construcción de la propuesta de solución. Las historias de usuarios permitieron administrar los requisitos funcionales sin tener que elaborar gran cantidad de documentos formales y mantener una relación cercana con el cliente.

# Capítulo 3: Construcción y validación de la propuesta de solución

## 3.1 Introducción

En el presente capítulo se describe la implementación del *software*, fase donde se materializa el producto y cumple con los requisitos obtenidos al inicio de la investigación. Se definen los estándares de codificación que debe cumplir el equipo de desarrollo, así como los métodos y técnicas para la realización de las pruebas.

## 3.2 Diagrama de despliegue

El diagrama de despliegue se utiliza para mostrar la estructura física del sistema, incluyendo las relaciones entre el *hardware* y el *software* que se despliega, estas relaciones son representadas por los protocolos de comunicación que se utilizan para acceder a cada uno. En la Figura 9 se puede visualizar el diagrama de despliegue definido para la solución propuesta:

- **PC desarrollador:** computadora donde el desarrollador realiza los cambios a los paquetes.
- **Servidor de integración continua:** es el servidor donde se realiza todo el proceso y debe estar configurado para comprobar los cambios en el repositorio de control de versiones cada minuto. Por cada construcción el servidor comúnmente cuenta con reportes y un panel de control para mostrar la salud del *software*.
- **Servidor de control de versiones:** registra los cambios realizados en los paquetes a lo largo del tiempo, de modo que se pueda recuperar versiones específicas más adelante.
- **Repositorio:** mantiene todos los paquetes del proyecto en un lugar centralizado, donde los desarrolladores se conectan y descargan una copia local del proyecto, el desarrollador envía al repositorio periódicamente los cambios que se realizan y van actualizando su directorio de trabajo.

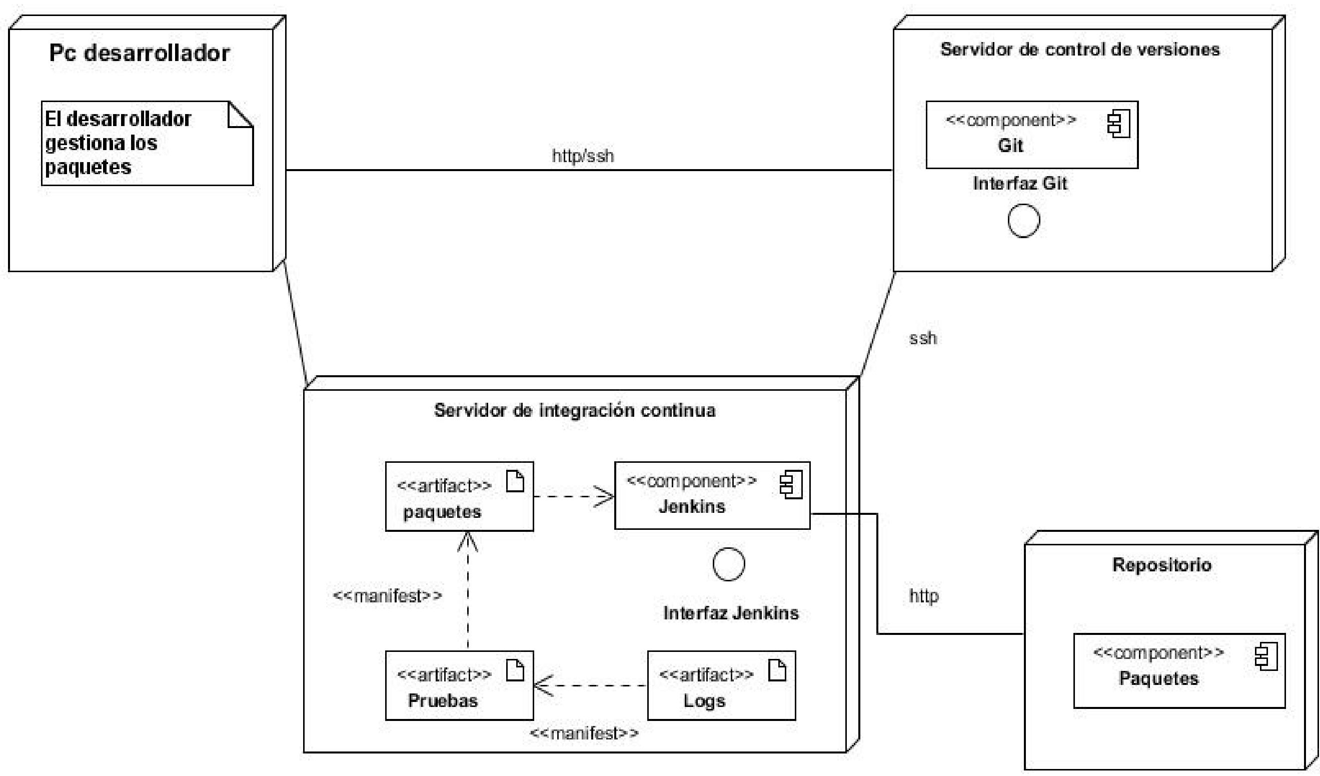



Figura 9: Diagrama de despliegue

*(Fuente: Elaboración propia)*

## 3.3 Estándares de codificación

Los estándares de codificación son reglas de codificación que permiten tener una programación homogénea, comprendiendo todos los aspectos de la generalización del código, pues la aplicación debe de estar implementada como si un único programador escribiera el código de una sola vez. La usabilidad de estos permite conservar el código fuente entendible y fácil de mantener, además de mejorar la forma en la que se programa [27]. Para el desarrollo de la solución se utilizaron los siguientes estándares de

codificación (ver Figura 10):

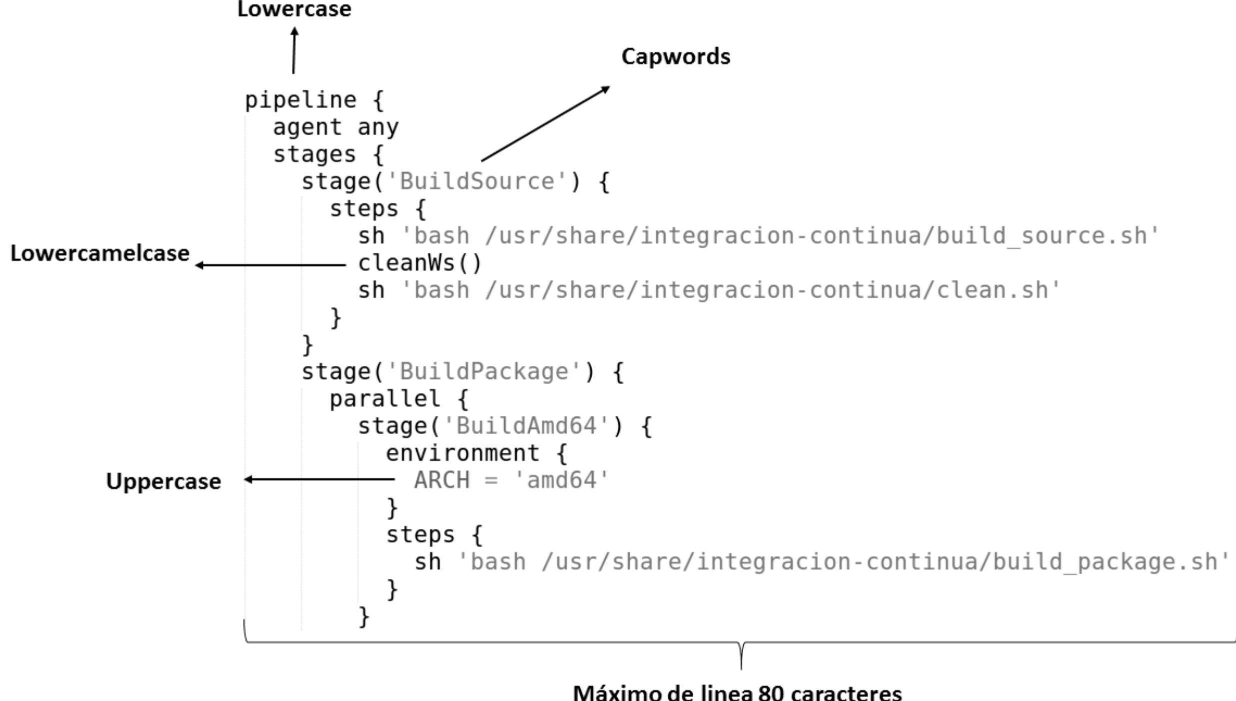



Figura 10: Estándares de codificación

*(Fuente: Elaboración propia)*

**Máxima longitud de las líneas**

- Se limitaron todas las líneas a un máximo de 80 caracteres.

**Convenciones de nombres**

- Se utilizó *Lowercase* (toda la palabra en minúscula) para las palabras claves.
- Se utilizó *Uppercase* (toda la palabra en mayúscula) para las variables de entorno.

- Se utilizó *CapWords* (palabras que comienzan con mayúsculas) para los nombres de las *stage* (etapas).
- Se utilizó *Lowercamelcase* (palabras que comienzan con minúsculas) para los nombres de los métodos.
- No se utilizaron los caracteres 'l' (letra ele minúscula), 'O' (letra o mayúscula) o 'I' (letra i mayúscula) como nombres de variables de un solo caracter debido a que, en algunas fuentes, estos caracteres son indistinguibles de los numerales uno y cero.

**Otras consideraciones**

- Se rodearon los operadores binarios con un espacio en cada lado.

## 3.4 Pruebas de *software*

Las pruebas constituyen “Una actividad en la cual un sistema o componente es ejecutado bajo condiciones específicas, se observan o almacenan los resultados y se realiza una evaluación de algún aspecto del sistema o componente” [28]. Se planifican y se ejecutan una serie de pruebas definidas por la metodología AUP variación UCI; las cuales son las pruebas internas y pruebas de aceptación.

### 3.4.1 Pruebas de internas

Para la realización de las pruebas internas, las cuales permiten verificar el resultado de la implementación. Las pruebas de unidad tienen como objetivo verificar la unidad más pequeña del diseño del *software*. Se concentran en la lógica del procesamiento interno y en las estructuras de datos tales como: código fuente, archivos binarios, archivos de datos, entre otros. Este tipo de prueba se puede aplicar en paralelo a varios componentes.

Las pruebas de unidad, denominada a veces prueba de caja de cristal, es un método de diseño de casos de prueba que usa la estructura de control del diseño procedimental para obtener los casos de prueba. La técnica de prueba de caja blanca utilizada en la investigación es el camino básico, que permite obtener

una medida de la complejidad lógica de un diseño procedimental y usar esa medida como guía para la definición de un conjunto básico de caminos de ejecución [23]. En la figura se muestra el procedimiento de la etapa de publicación de binarios.

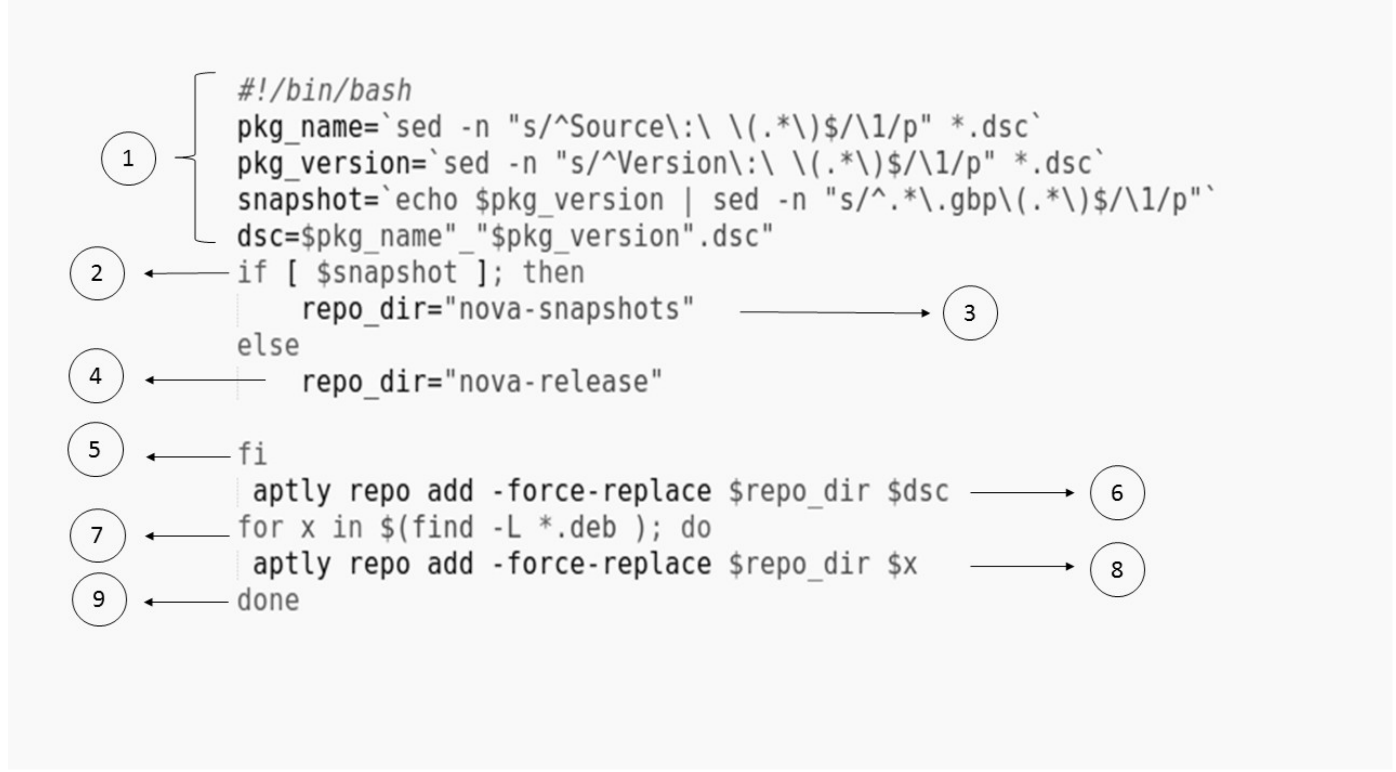



Figura 11: Procedimiento de la etapa publicación de binarios.

*(Fuente: Elaboración propia)*

### Grafo de flujo y cálculo de caminos linealmente independientes

Caminos linealmente independientes:

- Camino 1: 1,2,3,5,6,7,8,7,9

- Camino 2: 1,2,4,5,6,7,8,7,9
- Camino 3: 1,2,3,5,6,7,9

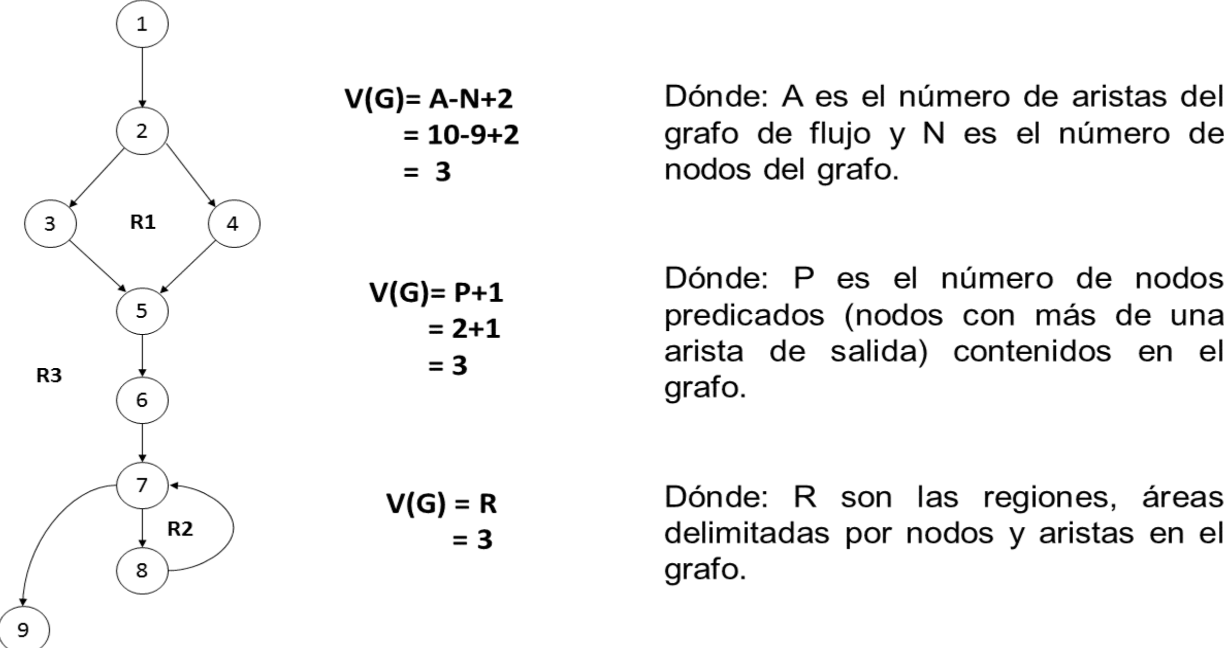



Figura 12: Grafo de flujo y cálculo de caminos linealmente independientes.

*(Fuente: Elaboración propia)*

**Diseño de caso de prueba para los caminos linealmente independientes**

*Tabla 10: Caso de prueba para el camino 1.*

| **Diseño de caso de prueba para el camino 1** | |
|---|---|
| Descripción | Creación de un snapshot. |
| Condición de ejecución | El desarrollador realizó cambios en el código sin realizar un reléase. |

| | |
|---|---|
| Entrada | Paquete: Hello.gbp~210245620.deb |
| Resultado esperado | Se publican los archivos en el repositorio nova-snapshot. |

*Tabla 11: Caso de prueba para el camino 2.*

| **Diseño de caso de prueba para el camino 2** | |
|---|---|
| Descripción | Creación de un release. |
| Condición de ejecución | El desarrollador realizó cambios en el código y se realizó un release. |
| Entrada | Paquete: Hello.1.0.dsc |
| Resultado esperado | Se publican los archivos en el repositorio nova-release. |

*Tabla 12: Caso de prueba para el camino 3.*

| **Diseño de caso de prueba para el camino 3** | |
|---|---|
| Descripción | Creación de un release o un snapshot. |
| Condición de ejecución | El desarrollador realizó cambios en el código ,pero no se publican en el repositorio. |
| Entrada | Sin paquete. |
| Resultado esperado | No publica artefactos en el repositorio. |

### 3.4.2 Pruebas de Aceptación

Independientemente de que un equipo ajeno al *software* realice el proceso de pruebas, es recomendable

que el cliente designe a personal que haga parte de los procesos de negocio para la ejecución de pruebas de aceptación, incluso que los usuarios finales participen en este proceso. Cuando las pruebas de aceptación son ejecutadas en instalaciones o ambientes proporcionados por los desarrolladores se les denominan pruebas Alpha, cuando son ejecutadas desde la infraestructura del cliente se les denomina pruebas Beta.

En los casos en que las pruebas de aceptación del producto se hayan ejecutado en el ambiente del proveedor, el aplicativo no podrá salir a producción, sin que se hayan ejecutados las respectivas pruebas Beta en el ambiente del cliente. Las pruebas Alpha son opcionales, pero las pruebas Beta son obligatorias [29]. Se realizaron pruebas de aceptación Alpha y Beta con el cliente el cual emitió un acta de aceptación de productos de trabajo (ver Anexo 3) en total conformidad con la solución desarrollada.

**Resultados de las pruebas**

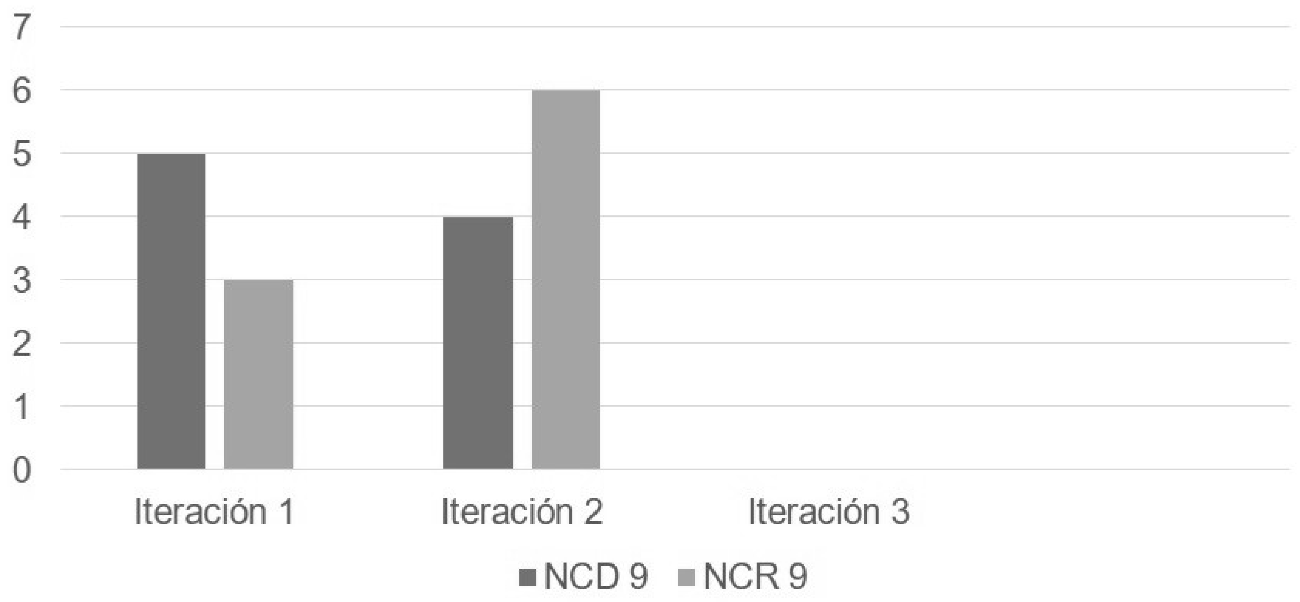



Figura 13: Resultado de las pruebas.

Fuente: (Elaboración propia)

En la primera iteración de prueba se encontraron cinco no conformidades.

- 1 NC: Dirección del repositorio incorrecta.
- 2 NC: Las bases de la herramienta sbuild no estaban creadas.
- 3 NC: La herramienta sbuild estaba desactualizada.
- 4 NC: La dirección del archivo Jenkinsfile era incorrecta.
- 5 NC: La herramienta no obtenía la rama pristine-tar del repositorio.

Fueron solucionas en la primera iteración tres NC.

- 1 NCR: Se corrigió la dirección del repositorio.
- 2 NCR: Se crearon las bases de sbuild.
- 3 NCR: Se descargó del repositorio la última versión de sbuild.

En la segunda iteración de prueba se detectaron cuatro no conformidades.

- 1 NC: La herramienta Piuparts estaba desactualizada.
- 2 NC: La herramienta Lintian estaba desactualizada.
- 3 NC: No se publican los artefactos en el repositorio.
- 4 NC: La herramienta aptly estaba desactualizada.

Fueron solucionas en la segunda iteración seis NC, cuatro de la segunda iteración y las dos pendientes de la primera iteración.

- 1 NCR: Se corrigió la dirección del archivo jenkinsfile.
- 2 NCR: Se configuró la herramienta Jenkins para la obtención de la rama pristine-tar.
- 3 NCR: Se descargó del repositorio la última versión de Piuparts.
- 4 NCR: Se descargó del repositorio la última versión de Lintian.
- 5 NCR: Se corrigió la implementación del script Publish.
- 6 NCR: Se descargó del repositorio la última versión de aptly.

En la tercera iteración de prueba no se detectaron no conformidades.

## 3.5 Técnica de índice de satisfacción grupal (IADOV)

La técnica de IADOV es utilizada para determinar el nivel de satisfacción individual y grupal de los usuarios a partir de una encuesta elaborada. Dicha encuesta (ver Anexo 1) fue aplicada a 10 especialistas de CESOL.

La técnica de IADOV constituye una vía indirecta para el estudio de la satisfacción, ya que los criterios que se utilizan se fundamentan en las relaciones que se establecen entre tres preguntas cerradas que se intercalan dentro de un cuestionario y cuya relación el sujeto desconoce. Estas tres preguntas se relacionan a través de lo que se denomina el "Cuadro Lógico de IADOV"

**Cuadro lógico de IADOV**

*Tabla 13: Caso de prueba para el camino 3.*

<table>
<tr><td rowspan="4">5. Luego de haber visto la automatización del proceso de construcción y publicación de paquetes (fuentes y binarios) con integración continua para la Distribución Cubana GNU/Linux Nova refleje en qué medida le gusta la solución desarrollada.</td><td colspan="9">2. ¿Considera usted correcta la forma en que se realiza el proceso de construcción y publicación de paquetes (fuentes y binarios) en la Distribución Cubana GNU/Linux Nova actualmente?</td></tr>
<tr><td colspan="3">No</td><td colspan="3">No sé</td><td colspan="3">Sí</td></tr>
<tr><td colspan="9">3. ¿Considera usted factible la implementación de integración y entrega continua en el proceso de construcción y publicación de paquetes (fuentes y binarios) en la Distribución Cubana GNU/Linux Nova?</td></tr>
<tr><td>Sí</td><td>No sé</td><td>No</td><td>Sí</td><td>No sé</td><td>No</td><td>Sí</td><td>No sé</td><td>No</td></tr>
<tr><td>Me gusta mucho</td><td>1</td><td>2</td><td>6</td><td>2</td><td>2</td><td>6</td><td>6</td><td>6</td><td>6</td></tr>
<tr><td>Me gusta más de lo que me disgusta</td><td>2</td><td>2</td><td>3</td><td>2</td><td>3</td><td>3</td><td>6</td><td>3</td><td>6</td></tr>
<tr><td>Me da lo mismo</td><td>3</td><td>3</td><td>3</td><td>3</td><td>3</td><td>3</td><td>3</td><td>3</td><td>3</td></tr>
</table>

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Me disgusta más 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 sé decir | 2 | 3 | 6 | 3 | 3 | 3 | 6 | 6 | 4 |

La forma de utilizar la tabla es la siguiente:

- Cada encuestado recibe una evaluación individual en dependencia de las respuestas que dé a las preguntas cerradas. Para facilitar el procesamiento posterior, en el diseño de la encuesta se debe tener en cuenta que a estas preguntas sólo se responda de la forma prevista en el cuadro lógico de IADOV.
- Las respuestas a las preguntas 2 y 3 pueden ser SI, NO, NO SÉ, y a las preguntas 4, "Me gusta mucho", "Me gusta más de lo que me disgusta", "Me da lo mismo", "Me disgusta más de lo que me gusta", "No me gusta nada", o "No sé qué decir".

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 14: 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 |
|---|---|

El índice de satisfacción grupal (ISG) se expresa en una escala numérica que va desde 1 (máxima satisfacción), hasta -1 (máxima insatisfacción). El ISG se calcula mediante la siguiente fórmula:

$$ISG = \frac{A(+1) + B(+0{,}5) + C(0) + D(-0{,}5) + E(-1)}{N}$$

$$ISG = [8(+1) + 2(+0.5)]/10$$

$$ISG = 0.9$$

Los valores de ISG que se encuentran comprendidos entre -1 y -0,5 indican insatisfacción; los comprendidos entre -0,49 y +0,49 evidencian contradicción y los que caen entre 0,5 y 1 indican que existe satisfacción.

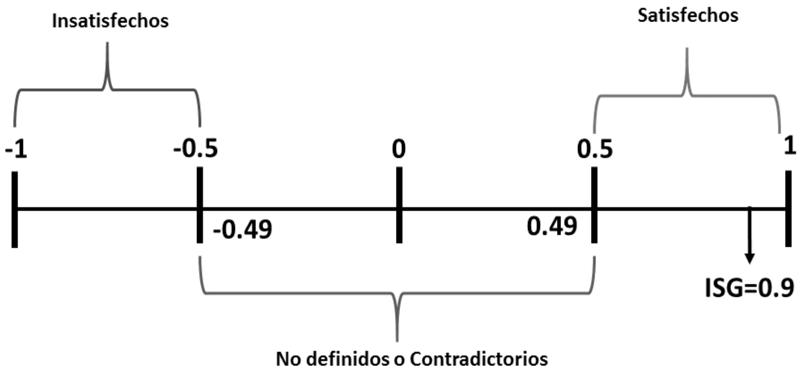



Figura 14: Ubicación del índice de satisfacción grupal.

*(Fuente: Elaboración propia)*

Mediante la técnica de IADOV se confirmó la factibilidad de uso del objetivo de la investigación,

expresando cuantitativamente en el alto ISG (0.9) y cualitativamente en los criterios emitidos en CESOL, lo que refleja la aceptación de la propuesta y el reconocimiento a su utilidad.

## 3.6 Resultados de la investigación

Una vez desarrollado el proceso de Integración continua y de haber validado la propuesta solución mediante las pruebas de software. Se concluye que el proceso de construcción y publicación de paquetes para la Distribución Cubana GNU/Linux Nova contribuyó a elevar el nivel de seguridad. La herramienta Jenkins permitió adicionar dos nuevas pruebas al proceso de construcción y publicación de paquetes, *Piuparts* y *Lintian*, garantizando que los paquetes cumplan las principales políticas de Debian, eliminando posibles vulnerabilidades y a su vez, se comprueba que cada paquete se instale, se modifique, se actualice y se elimine de forma correcta antes de la puesta en el repositorio. En la etapa de *ArchiveActefacts* se mantiene los artefactos en estado de listo en la carpeta de trabajo de Jenkins, posibilitando que el usuario pueda acceder a las últimas versiones de los paquetes en caso de pérdida de información.

El uso de la herramienta Git permite acceder al versionado de paquetes en el repositorio, lo cual permite mantener cada una de las versiones seguras y en caso de ser necesario se puede obtener cada una de las versiones si el paquete final es defectuoso o si este esta infestado por virus. El proceso solo podrá ser ejecutado por el rol administrador como único usuario, protegiendo el acceso a la herramienta con una contraseña, para así minimizar los riesgos de que un usuario inexperto interactúe con el proceso automatizado propuesto. El administrador de la herramienta será el único trabajador del centro que posea la llave digital para la firma de los paquetes antes de ser subidos al repositorio, ningún desarrollador del centro tendrá acceso a la llave digital a no ser que se realice una excepción para algún proyecto en específico. La automatización del proceso de construcción y publicación de paquetes con la herramienta de Integración continua Jenkins permitió elevar el nivel de seguridad eliminando cada uno de los problemas planteados en el inicio de la investigación.

## 3.7 Conclusiones parciales

Con la elaboración del diagrama de despliegue se obtuvo una mejor compresión de la estructura física del sistema, incluyendo las relaciones entre el hardware y el *software* que se despliega. La aplicación de los estándares de codificación, mejoraron la forma de programar y la conservación del código fuente entendible y fácil de mantener. Tras efectuar todas las pruebas previstas y haber alcanzado resultados satisfactorios, se concluyó que la solución cumple con las especificaciones dadas por el cliente, pues se lograron los objetivos propuestos. Además, se demostró que la solución implementada produce un resultado satisfactorio elevando el nivel de seguridad en la construcción y publicación de paquetes fuentes y binarios para la Distribución Cubana GNU/Linux Nova.

## Conclusiones generales

- El análisis a los diversos sistemas permitió conocer nuevas ideas y características relevantes en el proceso de integración continua, posibilitando la selección de *Jenkins* como herramienta más apropiada para el desarrollo de la solución.
- La aplicación de la metodología AUP variación para la UCI permitió estructurar, planear y guiar el proceso de desarrollo del *software*, facilitando la obtención y especificación de los requisitos, así como la elaboración de los artefactos propuestos y el diseño de la solución.
- Con la automatización del proceso de construcción y publicación de paquetes fuentes y binarios en la Distribución Cubana de GNU/Linux Nova, se logró elevar el nivel de seguridad mediante el uso de la herramienta de integración continua *Jenkins*.
- Una vez realizadas las pruebas previstas y haber alcanzado resultados satisfactorios, se logró garantizar que la solución cumpla con los requisitos establecidos y las necesidades del cliente.

# Recomendaciones

- Adicionar compilación distribuida al proceso de integración continua.
- Adicionar una nueva etapa en el archivo *jenkinsfile* y crear un nuevo *script bash* donde se defina el uso de un antivirus de los definidos por el centro o alguna otra herramienta de protección que disponga el proyecto, para velar por la seguridad de los paquetes en cuanto a protección contra *malware* o virus.

## Bibliografía

1. **UCI.** Universidad de las Ciencias Informáticas [En línea] Misión, 2014. Citado el: 15 de noviembre del 2017]. Disponible en: http://www.uci.cu/universidad/mision.
2. **PIERRA FUENTES, Allan**. Nova, distribución cubana de GNU/Linux : reestructuración estratégica de su proceso de desarrollo. Tesis de maestría. Universidad de la Ciencias Informáticas. La Habana, 2011.
3. **ALBERT, José.** Paquetes en DEBIAN – Parte I (Paquetes, Repositorios y Gestores de Paquetes.) [En línea] Desde Linux, 2016. [Citado el: 8 de diciembre del 2017]. Disponible en: https://blog.desdelinux.net/estudio-de-los-paquetes-de-debian-parte/.
4. **FERNANDEZ SANGUINO, J.** The Debian GNU/Linux FAQ - Basics of the Debian package. [En línea] *The Debian GNU/Linux FAQ - Basics of the Debian package*, 2011. [Citado el: 15 de diciembre del 2017]. Disponible en: http://www.debian.org/doc/manuals/debian-faq/ch-.
5. **SCHWARZ, Ian Jackson and Christian.** *Debian Policy Manual*. [En línea] *Debian Policy Manual,* 1998. [Citado el: 4 de febrero del 2018]. Disponible en: https://www.debian.org/doc/debian-policy/#document-index.
6. **GIT.** Git*.* [En línea] *Git ,* 2014. [Citado el: 21 de enero del 2018]. Disponible en: https://git-scm.com/book/es/v2.
7. **FOWLER, MARTIN. 2006.** *Continuous Integration*. [En línea] *Continuous Integration,* 1 de Mayo de 2006. [Citado el: 28 de marzo del 2018]. Disponible en: http://martinfowler.com/articles/continuousIntegration.html.
8. **GARZAS, Javier.** Aprende a implantar integración continua desde cero (I). [En línea] JavierGarzas.com, 2014. [Citado el: 20 de diciembre del 2017]. Disponible en: http://www./2014/08/implantar¬integracion¬continua.html.
9. **HINOSTROZA MISAEL, Oscar.** Incorporación de la Integración Continua en el Desarrollo de *software*, Noviembre de 2011. [22 de diciembre del 2018].
10. **JEZ, Humble; FARLEY, David.** *Continuous Delivery.* 2011. [Citado el: 11 de mayo del 2018].

11. **ATLASSIAN,** *Atlassian Word. Comparison* bamboo-vs-jenkins*.* [En línea] *Atlassian Word. Comparison* bamboo-vs-jenkins*,* 2016. [Citado el: 20 de diciembre del 2017]. Disponible en: http://www.atlassian.com/software/bamboo/comparison/bamboo-vs-jenkins.
12. ***THE SLANT COMMUNITY*.** *The Slant Community*. [En línea] *Slant,* 2017. [Citado el: 12 de abril del 2018]. Disponible en: https://www.slant.co/topics/799/~best-continuous-integration-tools.
13. **ROME, José Enrique.** Integración y Despliegue Continuo: Monitorización. Integración y Despliegue Continuo: Monitorización. Sevilla: s.n., 2016. [Citado el: 15 de febrero del 2018].
14. **ROMERO BERSABÉ, José Enrique.** Integración y Despliegue Continuo: Monitorización de y automatización de pruebas *software.* Sevilla: s.n., 2016. [Citado el: 18 de marzo del 2018].
15. **LABORATORIO NACIONAL DE CALIDAD DEL SOFTWARE.** Ingeniería del *software*: Metodologías y ciclos de vida. Marzo 2009. [Citado el: 14 de octubre del 2017].
16. **SÁNCHEZ RODRÍGUEZ, Tamara.** Metodología de desarrollo para la actividad productiva de la UCI. Metodología de desarrollo para la actividad productiva de la UCI, 3 de junio de 2015. [Citado el: 13 de abril del 2018].
17. **FÍRVIDA DOMÉSTEVEZ, Abel; RODRÍGUEZ PINO, Adisleydis.** Guano, entorno de escritorio cubano, libre y de código abierto. Tesis de pregrado. Universidad de las Ciencias Informáticas. La Habana, 2009. [17 de febrero del 2018].
18. **GENBETADEV.** Mejora tu código usando java. [En línea] Mejora tu código usando java, 2016. [Citado el: 10 de enero del 2018]. Disponible en: https://www.genbetadev.com/java-j2ee/mejora-tu-codigo-java-usando-groovy
19. **RUÍZ PÉREZ, Andrés.** Implementación del proyecto “ArduPlane” en el servidor de integración continua del GARP. 19 de junio de 2015. [Citado el: 15 de febrero del 2018].
20. **DEP 14.** Deb 14 [En línea] Deb 14, 2014 [Citado el: 2 de mayo del 2018]. Disponible en: https://deb.debian.net.
21. **VISUAL PARADIGM FOR UML.** Libere las revisiones de la transferencia directa y del *software* | CNET. [En línea]. [Citado el: 28 de noviembre del 2017]. Disponible en:

http://descargar.cnet.com/Visual-Paradigm-for-UML/3000-2247_4-42700.html

22. **LARMAN. Craig.** Applying UML and patterns. Introduction to object-oriented analysis and design and the united process. 2002. 616 p. [Citado el: 11 de mayo del 2018].
23. **PRESSMAN, Roger S.** Métodos convencionales para la ingeniería del *software*. En: Darrel Ince. Ingeniería del *software* un enfoque práctico. México: 2001. Parte III, 258 p. [Citado el: 18 de marzo del 2018].
24. **SOMERVILLE, Ian.** *software* engineering. AddisonWesley Pub Co, 6ta edición, Agosto 2000. [Citado el: 17 de enero del 2018].
25. **JEFFRIES, Ron; ANDERSON, Ann; HENDRI, Chet.** Extreme Programming Installed. s.l.: Addison-Wesley Professional, 2001. [Citado el: 8 de marzo del 2018].
26. **BASS, L.; CLEMENTS, P.; KAZMAN, R.** *software* Architecture in Practice, 2nd Edition, Addison Wesley, 2003. [Citado el: 4 de febrero del 2018].
27. **ARIAS CALLEJA, Manuel.** Estándares de codificación, 2015. [Citado el: 2 de marzo del 2018].
28. **IEEE.** Std 829-1998. “Standard for *software* Test Documentation”. [Citado el: 4 de noviembre del 2017].
29. **ZAPATA S., Javier.** Pruebas de software. [En línea] *Pruebas de software,* 2013. [Citado el: 18 de noviembre del 2017]. Disponible en: https://pruebasdel*software*.wordpress.com/.
30. **HOME- CENTERS FOR MEDICARE & MEDICAID SEVICES.** Home- Centers for Medicare & Medicaid Sevices. [En línea] *Home- Centers for Medicare & Medicaid Sevices.* [Citado el: 16 de enero del 2018]. Disponible en: https://www.cms.gov/Research-Statistics-Data-and-Systems/CMS-Information-Technology/XLC/Downloads/Select.
31. **INFORMÁTICA, LA OFICINA DE PROYECTOS.** pmoinformatica.com. [En línea] *pmoinformatica.com,* 2018. [Citado el: 11 de enero del 2018]. Disponible en: http://www.pmoinformatica.com/p/pruebas-de-*software*.html.
32. **RAI, P.; MADHURIMA, DHIR, S.; MADHULIKA, & GARG, A.** *2nd International Conference on Computing for Sustainable Global Development.* New Delhi, India: IEEE: s.n., 2015. [Citado el: 11

de octubre del 2017].

33. **SAAVEDRA GUTIERREZ, Jorge A.** El mundo informático. [En línea] El mundo informático, 5 de mayo de 2007. [Citado el: 19 de mayo del 2018]. Disponible en: https://jorgesaavedra.wordpress.com/2007/05/05/lenguajes-de-programacion/.
34. **SHAHIN, M.; BABAR, A., & ZHU, L.** *Continuous Integration*, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. IEEE Software. *Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. IEEE software.* 2017. [Citado el: 4 de diciembre del 2017].
35. **SONI, M.** *End to End Automation On Cloud with Build pipeline: The case for DevOps in Insurance industry. IEEE International Conference on Cloud Computing in Emerging Markets,* 2015. [Citado el: 10 de diciembre del 2017].

# Anexo 1

Encuesta realizada a los especialistas correspondiente a la técnica de IADOV:

Especialista, le invito a responder el siguiente cuestionario con el fin de conseguir su colaboración en la presente investigación, solicito que exprese en sus repuestos criterios verídicos que guíen al autor del trabajo. Marque en cada pregunta con una X en una sola opción y en el caso de la 5 responda brevemente. Por el tiempo brindado, muchas gracias.

1. ¿Considera que es necesario el proceso de construcción y publicación de paquetes fuentes y binarios para la Distribución Cubana GNU/Linux Nova?

Sí ___ No ___ No sé___

2. ¿Considera usted correcta la forma en que se realiza el proceso de construcción y publicación de paquetes fuentes y binarios en la Distribución Cubana GNU/Linux Nova actualmente?

Sí ___ No ___ No sé___

3. ¿Considera usted factible la implementación de integración y entrega continua en el proceso de construcción y publicación de paquetes fuentes y binarios en la Distribución Cubana GNU/Linux Nova?

Sí ___ No ___ No sé___

4. Luego de haber visto la automatización del proceso de construcción y publicación de paquetes fuentes y binarios con integración continua para la Distribución Cubana GNU/Linux Nova refleje en qué medida le gusta la solución desarrollada.

__Me gusta mucho __Me disgusta más de lo que me gusta __Me gusta más de lo que me disgusta
__No me gusta nada __Me da lo mismo __No sé decir

5. ¿Qué opina usted acerca de los beneficios que traería para el centro CESOL disponer de una solución que automatice el proceso de construcción y publicación de paquetes fuentes y binarios para la Distribución Cubana GNU/Linux Nova?

## Anexo 2

Entrevista realizada a los especialistas del centro.

Preguntas:

- ¿Qué es un paquete fuente?
- ¿Qué es un paquete binario?
- ¿Qué tipos de pruebas realizan a los paquetes?
- ¿Qué son las dependencias de paquetes?
- ¿Cuáles son las arquitecturas más usadas en el centro?
- ¿Qué es un controlador de versiones?
- ¿Qué controlador de versiones utilizan para monitorear las versiones de los paquetes?
- ¿Cómo se realiza el proceso de construcción y publicación de paquetes fuentes y binarios en el centro CESOL?
- ¿Qué es integración continua?
- ¿Cuál es la metodología definida por el centro para guiar el desarrollo del *software*?

## Anexo 3

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 **Manuel Alejandro Ricardo Serrano** de la Facultad 1 de la Universidad de las Ciencias Informáticas y en función de la ejecución del proyecto: **Herramienta de Integración continua con Jenkins**, se hace entrega del producto que se relaciona a continuación:

- **Herramienta de Integración continua con Jenkins**

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:** Manuel Alejandro Ricardo Serrano | **Nombre y apellidos:** Yoandy Pérez Villazón |
| **Cargo:** Estudiante Facultad 1 | **Cargo:** Director de CESOL |
| **Firma:** | **Firma:** |

Universidad de las Ciencias Informáticas
Centro CESOL Facultad 1

**Fecha:** 10/05/2018