Universidad de las Ciencias Informáticas
Facultad 10

TRABAJO DE DIPLOMA
PARA OPTAR POR EL TÍTULO DE
INGENIERO EN CIENCIAS INFORMÁTICAS

Título: Sistema para automatizar la generación de paquetes binarios de la distribución Nova

Autora: Mónica Mª Albo Castro

Tutor: Ing. Anielkis Herrera González
Co-Tutor: Msc. Leonardo Herrera Boza

Ciudad de la Habana
Junio/ 2008

DECLARACIÓN DE AUTORÍA

Declaro ser autora de la presente tesis y reconozco a la Universidad de las Ciencias Informáticas los derechos patrimoniales de la misma, con carácter exclusivo.

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

Mónica Ma. Albo Castro	Anielkis Herrera Gonzáles
_____	_____
Firma del Autor	Firma del Tutor

Datos de Contacto

Tutor: Ing. Anielkis Herrera González

- Graduado en la Universidad de Ciencias Informáticas en el curso 2005-2006.
- Se desempeña como profesor universitario.
- Ha trabajado en los proyectos SAFRE y Nova en el cual se desempeña actualmente.

Co-Tutor: MSc. Leonardo Herrera Boza

- Graduado en el Instituto Superior Pedagógico “Rafael María de Mendive” en el curso 2000-2001.
- Se graduó de Máster en Ciencias en diciembre del 2007.
- Se desempeña como Jefe de Departamento de Enseñanza Asistida por Computadoras, Dirección de Teleformación.
- Ha participado en proyectos productivos nacionales e internacionales como especialista en temáticas de Teleformación.
- Posee publicaciones en revistas referenciadas.

Agradecimientos

A la Revolución Cubana y al compañero Fidel por darme la posibilidad de formarme como profesional de la vanguardia.

A mis padres por su apoyo incondicional en todo momento y por haberme enseñado el camino a seguir para llegar a donde estoy hoy.

A mis familiares, los que aun están presentes y los que ya no están, por ser parte también de mi formación como persona de bien con aspiraciones de ser cada día mejor.

Al tutor el Ingeniero Anielkis Herrera González y el co-tutor el Máster en Ciencias Leonardo Herrera Boza de esta investigación por ser guías incondicionales.

A mi compañero y amigo que siempre ha estado ahí en los momentos de flaqueza para darme las fuerzas necesarias para seguir, Joelsy Porven Rubier.

A mis amigos, los que han estado en los momentos que los he necesitado.

A todos aquellos que de una forma u otra han tenido que ver con mi formación profesional y como una persona cada vez mejor.

Resumen

Actualmente el equipo de desarrollo de la distribución NOVA trabaja para lograr que ésta sea más que una personalización. Para ello se investiga en las características de la misma que aun representan una desventaja para el uso de la misma por los usuarios. Una de éstas es el proceso de gestión de paquetes, donde para instalar un nuevo software necesita compilarlo desde los paquetes de fuentes, haciendo este proceso engorroso, además de no ser recomendable para usuarios comunes. La realización de este trabajo de forma manual no se considera óptimo de la gestión de paquetes, pues requiere de conocimientos profundos de computación. Esta situación fue lo que motivó la presente investigación dirigida a la modelación de una herramienta para la generación de paquetes de código binario para un repositorio, contribuyendo a optimizar el proceso de instalación de aplicaciones de la distribución Nova. Para ello se describen los conceptos fundamentales relacionados con la investigación y los artefactos fundamentales que son necesarios para el diseño de la herramienta. Aplicando las metodologías estudiadas se logra establecer el diseño de una herramienta que permitirá a la distribución Nova generar sus propios paquetes binarios a partir de los diversos repositorios de fuentes.

PALABRAS CLAVE

Paquete de software, proceso de compilación.

Tabla de contenido

Introducción

El avance de las tecnologías informáticas ha devenido en el desarrollo de lo que se conoce actualmente como sociedad de la información. La misma no es más que una comunidad que basa su desarrollo en la gestión de la información y el conocimiento. Actualmente se debate en el dilema de la privatización o liberación del conocimiento, en nuestro ámbito, del software.

En los inicios de la creación de software se compartía todo ese conocimiento dentro de la comunidad vinculada a éstos. En la década del 80' las nuevas empresas de la rama comenzaron a obstaculizar la cooperación entre desarrolladores [Stallman, R., 2007].

En 1984, Richard Stallman, desarrollador de software y defensor del software libre, creó el proyecto **GNU** con la idea de crear un sistema operativo siguiendo los **estándares de UNIX**. Los primeros pasos de este proyecto fueron la elaboración de varias herramientas de trabajo entre ellas: GNU Emacs como un editor de texto que podía interpretar algunos lenguajes, el compilador GCC y X Window como sistema de ventanas, entre otras que se incorporarían al sistema [Stallman, R., 2007].

Lo más importante es que un sistema operativo posee un sistema central, más conocido como **kernel** (núcleo), el cual es más complejo de elaborar. Durante los primeros años de desarrollo se probaron varios núcleos de sistemas pero ninguno logró satisfacer las necesidades del sistema GNU que se estaba desarrollando [Stallman, R., 2007].

En 1991 sale a la luz un nuevo kernel que marca un precedente en el desarrollo del software libre y de los sistemas operativos. Este kernel se comenzó a desarrollar como un hobby de Linus Torvalds y fue denominado **Linux**. Se distribuyó bajo la **Licencia Pública General (GPL)** de GNU [Linux Onlines Inc, 2007a].

Cuando surgió Linux se hizo la prueba con este kernel y junto a las herramientas ya creadas por el proyecto. El resultado de ello fue el sistema operativo **GNU/Linux**, más conocido como Linux, actualmente utilizado por millones de personas [Free Software Foundation, 2007]. Este es un sistema operativo totalmente libre y funcional compatible con **UNIX**, desarrollado por una comunidad libre. Surge además en busca de una alternativa al software propietario que crea abismos de colaboración entre los

desarrolladores de software.

GNU/Linux comenzó a ser optimizado con distintas opciones en las diferentes universidades técnicas y compañías, y a distribuir estas versiones. Esta situación ha devenido en lo que se conoce como distribuciones de GNU/Linux que no son más que versiones del sistema original [Linux Onlines Inc, 2007b]. Estas distribuciones (abreviada a *distros*) se diferencian, entre otras muchas características, por las herramientas de configuración y los sistemas de administración de paquetes de software para la instalación que incluyen [Wikipedia, 2007].

Un paquete de software es un conjunto de ficheros que forman parte de un programa, es decir, contienen el código de éste. Pueden ser de código binario o de código fuente. Los de código binario son aquellos que luego de ser compilados poseen estructura determinada por la distribución que lo genere, por lo que ya están en lenguaje máquina. Los de código fuente son los que están en un lenguaje de programación, en el cual fue escrito el programa.

En Cuba se ha iniciado hace unos años un proceso de migración dentro del cual se contempla el desarrollo de una distribución GNU/Linux propia. En la Universidad de Ciencias Informáticas (UCI) en febrero del 2003 surgió un equipo de desarrollo para la creación de una distribución GNU/Linux denominada Nova. El equipo de desarrollo de Nova, tomó como base a la distribución **Gentoo Linux**, aunque actualmente trabaja en función de llevarla más allá de una personalización [Goñi, A., 2007].

Gentoo Linux es una distribución que no está ideada para usuarios con poco conocimiento sobre el tema. Es una distribución que tiene una gran adaptabilidad y su sistema gestor de paquetes, **Portage**, es considerado, en la mayoría de los casos, como uno de los más robustos [Herrera, A. y Rodríguez, Y., 2006].

El equipo de desarrollo de la distribución Nova pretende lograr una plataforma de desarrollo y que además pueda ser usada fácilmente por usuarios sin experiencia informática. Para ello aprovecha las ventajas que le brinda Gentoo y mejora las desventajas que esta distribución pueda acarrear. Entre las ventajas que Nova hereda de la distribución tomada como base, se destaca Portage, el cual aunque está diseñado para el trabajo con paquetes de fuentes, permite instalar paquetes de binarios de otras distribuciones.

El trabajo con paquetes de fuentes de Portage implica un **tiempo de compilación**. Esto representa un problema a la hora de instalar nuevos paquetes, fundamentalmente por usuarios con escasos conocimientos sobre Linux. Solucionar la dificultad que representa el tiempo de compilación es de especial importancia para lograr una distribución de propósito general.

Un grupo del equipo de desarrollo comenzó a trabajar en esta dirección definiéndose como mejor solución la generación de un **repositorio** de binarios. Este debido a la necesidad de desarrollar otras tareas más prioritarias no se pudo expandir más allá y solo quedó con aplicaciones específicas [Burjans, L., 2007].

Actualmente la distribución NOVA para instalar un nuevo software necesita compilarlo desde los paquetes de fuentes, haciendo este proceso engorroso, además de no ser recomendable para usuarios comunes. La realización de este trabajo de forma manual no se considera óptimo de la gestión de paquetes, pues requiere de conocimientos profundos de computación. Atendiendo a la situación problémica planteada se ha delimitado el siguiente problema: La distribución Nova no cuenta con una herramienta que genere paquetes de código binario contribuyendo a la optimización del proceso de instalación de las aplicaciones.

La investigación se plantea como objeto de estudio la gestión de paquetes de código binario para la distribución Nova y como campo de acción el proceso de generación de dichos paquetes. Partiendo de la idea a defender que la existencia de una herramienta que genere los paquetes de código binario para un repositorio contribuirá a optimizar el proceso de instalación de aplicaciones de la distribución Nova.

Se propone como objetivo modelar una herramienta para la generación de paquetes de código binario para un repositorio, contribuyendo a optimizar el proceso de instalación de aplicaciones de la distribución Nova. En el transcurso de la misma serán realizadas diferentes tareas entre ellas:

- Sistematizar las tendencias más actuales en el uso de los repositorios de paquetes de código binario.
- Evaluar las metodologías y los lenguajes de programación existentes para el desarrollo de software libre.
- Modelar una herramienta para la gestión del proceso de generación de paquetes binarios.

Para el desarrollo de esta investigación se hará uso de métodos teóricos y métodos empíricos. Los

teóricos a utilizar son:

- Análisis-Síntesis para el análisis de la bibliografía sobre los diferentes enfoques de la gestión de paquetes de software y los sistemas de generación de paquetes binarios.
- Modelación que va a permitir una visión general de la gestión de paquetes binarios de la distribución Nova.

Los métodos empíricos que se van a usar son:

- Experimental el cual va a dar la posibilidad de que una vez creadas las condiciones necesarias se lleve a cabo la verificación del modelo mencionado anteriormente.
- Observación: para constatar la funcionalidad del desarrollo de una herramienta para la generación de paquetes binarios y la gestión de los mismos.

El documento de esta investigación esta estructurado en tres capítulos en cada uno de los cuales se desarrollará una sección importante del trabajo.

- El capítulo 1 comenzará con una fundamentación teórica sobre la generación de paquetes de código binario y los gestores de paquetes en las distribuciones GNU/Linux. Definiendo conceptos importantes así como la metodología de desarrollo a aplicar, lenguajes de programación y herramientas a emplear durante el proceso de realización.
- El capítulo 2 se centra un poco más en la descripción de la herramienta y en el inicio de la aplicación de la metodología de desarrollo a utilizar. Para ello se definen los requerimientos que debe cumplir y se lleva a cabo un análisis de los mismos describiendo los planes de entregas.
- El capítulo 3 muestra el diseño de la herramienta a desarrollar, describiendo la arquitectura y las funcionalidades básicas a implementar para satisfacer los objetivos.

Capítulo 1: Sistemas para la gestión de paquetes en GNU/Linux

La revolución de la informática y la computación se basa en el uso de las computadoras para la automatización de los procesos industriales y económicos. Para facilitar el trabajo con las mismas surgen los sistemas operativos, que ya en la actualidad permiten que hasta un niño pueda manejarlas.

Los sistemas operativos son el programa fundamental de una computadora, que controla los recursos y sirve como base al resto de los programas de aplicación [Tanenbaum, A, 1991]. En la actualidad existen varios sistemas operativos, cada uno con características particulares para llevar a cabo sus funciones y su distribución puede estar asociada al software libre o propietario. Entre los más conocidos dentro del ambiente de software propietario se encuentra Microsoft Windows y como sistema insignia del movimiento de software libre las distribuciones GNU/Linux. Estas últimas basadas en el núcleo Linux, las cuales gestionan los programas en forma de paquetes de software.

En este capítulo se expondrá que se entiende por paquete de software, las ventajas que presentan los de código binario sobre los de código fuente y como son gestionados en las distintas distribuciones GNU/Linux. Entre las múltiples metodologías de desarrollo de software y lenguajes de programación que existen se da una panorámica sobre los más utilizados y convenientes para el objetivo de la investigación. Por último, se presentan las tendencias actuales en cuanto a la gestión de paquetes, fundamentalmente la generación de paquetes de código binario.

1.1 Los paquetes de código binario

En la actualidad existen varios conceptos sobre este término que no se limitan a los paquetes de código binario o fuente. La Wikipedia lo asocia a una serie de programas que se distribuyen conjuntamente, ya sea porque sus funciones se complementan o por intereses de distribución comercial [Wikipedia, 2008a].

En el artículo “¿Qué son los paquetes de software?”, Sandoval coincide con el concepto anterior, aunque aporta una nueva definición. En la misma describe a los paquetes de software como un grupo de archivos de código fuente o binario conjuntamente con archivos de instrucciones sobre qué hacer, todos ellos comprimidos [Sandoval, A., 2006].

El diccionario informático de Alegsa ofrece dos conceptos. Uno de ellos se refiere a los paquetes de software como conjunto de archivos necesarios para la ejecución de un programa o para agregar características a un programa previamente instalado. El otro concepto que brinda este diccionario es vinculado a la programación orientada a objetos para nombrar a un grupo de clases relacionadas de un programa [Alegsa, 2008a].

La enciclopedia online Webopedia define que el término paquete de software en ocasiones se refiere al software en sí. Más específicamente un directorio presentado como un simple archivo que contiene todo lo necesario para su instalación, incluido el software. Además en esta enciclopedia se presenta un concepto similar al que expone la Wikipedia [Webopedia, 2008]. De las definiciones vistas hasta el momento, según opinión de la autora se puede concluir en tres puntos de vistas generales del término.

Uno sería el que refiere la enciclopedia Wikipedia, otro el expuesto en el diccionario informático de Alegsa con respecto la programación orientada a objetos. Atendiendo al enfoque desde el cual se tratan los paquetes de software se adopta el tercer punto de vista, en este se resumen varios de los conceptos referenciados. Se plantea que un paquete de software es un archivo comprimido que contiene ficheros con el código del software y otros con las indicaciones de cómo instalar el software.

Estos archivos tienen una estructura que está definida por el sistema operativo para el cual fue creado. Puede ser establecida a través de un formato estandarizado para que otro programa del sistema lo instale o como un instalador, es decir, que es capaz de instalarse a sí mismo [Alegsa, 2008a].

El último es el más usado por los sistemas propietarios y de código cerrado como Microsoft Windows, conociéndoseles como instaladores. Los formatos estandarizados son muy comunes en el software libre, sin embargo las distribuciones GNU/Linux, utilizan diferentes formatos para estructurar los paquetes de software.

En el concepto definido para la investigación los paquetes de software incluyen los ficheros con el código del programa. Los mismos pueden contener el código escrito por lo programadores que crearon el software o en código binario, también conocido como **lenguaje máquina** [Sandoval, A., 2006]. Ello se conoce en el ambiente del software libre como paquetes de fuentes o paquetes binarios. Los paquetes de fuentes son aquellos en los cuales los ficheros del software contienen el código que fue escrito por

quiénes lo desarrollaron.

La instalación de los paquetes de fuentes implica un proceso de compilación. Este consiste en la traducción del código del lenguaje de programación escrito por el programador en lenguaje binario o máquina, de modo que la computadora lo pueda entender. Con este proceso se crea un paquete optimizado para los componentes de la máquina. El paquete obtenido y que es instalado posteriormente es un paquete binario, es decir, el proceso de compilación lleva a cabo la transformación del código fuente en un código ejecutable por la computadora [ZonaSiete, 2004].

Los paquetes binarios poseen una estructura optimizada para un grupo de características de hardware, como la arquitectura. Por esta razón cada hardware requiere su propio paquete binario [ZonaSiete, 2004]. No obstante representa más ventajoso el trabajo con paquetes binarios ya que el proceso de compilación requiere de conocimientos profundos sobre programación.

Las distribuciones GNU/Linux más populares trabajan con paquetes binarios, aunque brindan la posibilidad a los desarrolladores de trabajar con paquetes de fuentes. Para ello utilizan servidores (repositorios) donde se publican los paquetes para las distintas arquitecturas, aunque existen paquetes binarios optimizados para cualquier arquitectura. De esta manera el usuario solo necesita tener instalado un gestor de paquetes con acceso a un repositorio para instalar un software. Estos repositorios no son más que un servidor o dispositivo en el cual se almacenan programas o archivos, en este caso los paquetes de software [Definición, 2008a].

1.2 Gestores de paquetes

Las distribuciones GNU/Linux, distribuyen los software en paquetes los cuales no se instalan de forma autosuficiente como los programas de Windows [Alegsa, 2008b]. El proceso de instalación, actualización o eliminación de estos paquetes puede realizarse manualmente, si se tienen conocimientos suficientes, o a través de los gestores de paquetes.

Las enciclopedias especializadas [Alegsa, 2008b; Wikipedia, 2008b] coinciden en cuanto al concepto de este término como un conjunto de herramientas que se encarga de automatizar los procesos de instalación, configuración, actualización y eliminación de los paquetes de software. A diferencia de otros

sistemas de administración de aplicaciones, éstos contendrán una base de datos para todas las librerías necesarias y se encargaran de conectar cada aplicación durante su instalación con la que le sea necesaria.

Los gestores de paquetes además de gestionar los paquetes realizan otras funciones dentro del sistema. En la mayor parte de la bibliografía solo se hace referencia a su función principal expuesta en el concepto. Otras de las funciones definidas por otros autores [Alegsa, 2008b; Wikipedia, 2008b] son:

- Comprobación de la suma de verificación (**checksum**), para evitar diferencias entre versiones locales y oficiales.
- Comprobación de la **firma digital**, para verificar la autenticidad del paquete.
- Resolución de **dependencias** para evitar problemas en el funcionamiento posterior del software.
- Gestionar actualizaciones, permitiendo la instalación de un software mas optimizado.
- Ordenar y agrupar los paquetes según la función principal que realizan facilitando con ello el trabajo de instalación y mantenimiento.

Las distintas distribuciones GNU/Linux pueden trabajar tanto con paquetes de fuentes como binarios, aunque en realidad depende de las opciones que brinden sus gestores de paquetes. Existen gestores de paquetes con la capacidad de manipular tanto paquetes de fuentes como binarios, pero no con las mismas opciones. Esto hace que sean caracterizados por gestionar el tipo de paquete al que más opciones le brindan; ejemplos de estos gestores tenemos RPM de Red Hat y DPKG/APT de Debian.

Los gestores de paquetes fuentes tienen como principal ventaja la posibilidad de lograr, cuando se compila, un software mejor optimizado, con la habilitación de las opciones deseadas solamente. Con la gestión de estos paquetes fuentes una desventaja es que la compatibilidad no es generalizada, lo que siempre implicará tener que compilar nuevamente en otro equipo. Aunque lo que se considera la desventaja fundamental y que genera cierto rechazo, en el caso de los usuarios finales, es el **tiempo de compilación** que suele ser bastante largo.

La principal ventaja de la gestión de paquetes binarios es precisamente que solo hay que instalar el paquete o en el mejor de los casos ejecutarlo. Aunque presentan la desventaja de tener paquetes de software optimizados. Poseen la ventaja de poder ser utilizados en cualquier equipo pues son compilados

con opciones generales.

Las distribuciones GNU/Linux se diferencias entre sí, además por el formato que emplean para estandarizar sus paquetes, ya sean binarios o fuentes, por lo que se han creado distintos gestores para gestionar cada uno de esos formatos. De las múltiples distribuciones conocidas hay un reducido grupo que se consideran primarias, dado que tomaron como base de su desarrollo solamente el sistema operativo GNU/Linux. Por tanto sus gestores también se consideran básicos, ya que en la mayoría de los casos son los que usan las distribuciones derivadas de ellas.

Entre estos formatos de paquetes tenemos los .rpm de Red Hat, .deb de Debian, .pkg (pkgbuild) de Arch Linux, .tgz de Salckware y .ebuild de Gentoo. Es importante ver cómo se gestionan éstos en las distribuciones primarias que les dieron origen, para tener un conocimiento más profundo sobre el tema.

1.2.1 *RPM*

El gestor de paquetes actual de la distribución Red Hat Enterprise Linux es Red Hat Package Manager (RPM), además es el formato de sus paquetes. Desarrollado por Marc Ewing y Erik Troan está enfocado a mejorar las fallas de sus predecesores **RPP**, **PMS** y **PM** [Bayley, E. C., 2000]. RPM gestiona tanto paquetes de código fuente como paquetes de código binario, esto permite satisfacer a todo tipo de usuarios, puesto que los desarrolladores podrán compilar e instalar las aplicaciones creando las personalizaciones deseadas [Barnes, D., 1999].

Dentro de las opciones básicas que brinda RPM se encuentra la instalación, actualización, verificación y desinstalación o eliminación de paquetes de código binario. En el caso de los paquetes fuentes sus funciones fundamentales son instalación básica, dejando los fuentes instalados en el directorio /usr/src/redhat/sources y la compilación que crea los binarios correspondientes y los deja listos para su instalación [Fernández G., D., 2002].

En caso de que un paquete que estemos instalando necesite que exista otro ya instalado, el gestor abortará la instalación y nos mostrará un mensaje con las dependencias necesarias que deben estar instaladas. En una situación de desinstalación si existe un paquete instalado que dependa del que se está desinstalando, al igual que el caso anterior mostrará un mensaje de error con la información de la

situación.

Otra cuestión importante es cuando se actualizan paquetes ya instalados guarda el **fichero de configuración** de la versión anterior de la aplicación e instala el nuevo permitiendo al usuario mantener las personalizaciones realizadas con anterioridad [Red Hat Software Inc., 1999].

Los paquetes que gestiona RPM tienen como extensión .rpm, pueden ser fuentes o binarios, los primeros se generan con un formato especial. Contiendo el archivo compactado de la aplicación, un fichero con extensión .spec, que contiene una descripción de la aplicación y las indicaciones de cómo debe construirse. Además de los **parches** que se le pueden agregar para una correcta compilación e instalación. Los paquetes de código binario de RPM son un fichero binario que se puede dividir en 4 secciones:

- la inicial, que almacena la información del paquete, aunque en versiones más actuales, dada su inflexibilidad se usa lo que se conoce como **cabecera de la estructura**.
- la firma, que contiene la información necesaria para verificar la integridad y autenticidad del paquete.
- la cabecera, esta sección presenta toda la información disponible sobre el paquete.
- los ficheros de la aplicación, como bien dice el nombre, en esta parte del paquete se encuentran los ficheros de la aplicación compilados y compactados [Bailey, E. C., 2000].

1.2.2 *DPKG/APT-Aptitude*

La distribución Debian posee una herramienta de bajo nivel similar a RPM, Debian Package (DPKG). La misma necesita un sistema que sirva de interface de alto nivel, paro poder solventar conflictos complejos que se generan durante la gestión, como las dependencias [Wikipedia, 2008c].

La interface que necesitaba DPKG surge como APT (Advanced Packaging Tool), aunque en sí es solo una biblioteca de librerías de funciones de C++ que va a apoyar y simplificar dichos conflictos. Para esta herramienta se crearon interfaces más cómodas como el Aptitude, que tiene una interfaz de texto **NCURSES**, y Synaptic con una interfaz gráfica **GTK+**, entre otros [Wikipedia, 2008d]. Estas herramientas en conjunto conforman el gestor de paquetes de Debian, que puede gestionar tanto fuentes como binarios.

El gestor de paquetes dpkg empaqueta y desempaqueta los paquetes fuentes con la herramienta dpkg-source [Wikipedia, 2008c]. También con APT, permite descargar y compilar paquetes de fuentes usando la herramienta apt-get, para ello debe modificarse el fichero sources.list [Noronha, G., 2004]. Para el caso de los paquetes de código binario si tienen ambas herramientas una gran cúmulo de opciones, que van desde la instalación y desinstalación hasta el chequeo del estado de un paquete.

Los paquetes fuentes gestionados por este sistema tienen una estructura muy parecida al estándar Unix, es decir son en sí un fichero compactado cuyo contenido puede variar. Si el software es creado solamente para Debian, está compuesto por el fichero con formato .dsc que contiene la lista de los ficheros compactados y el checksum, además de un fichero compactado con los fuentes del software.

En el caso de un software creado independientemente de una distribución el paquete contendrá el fichero .dsc, un compactado origin con los fuentes originales y uno diff con las diferencias entre éste y los fuentes debianizados que estarán en otro compactado [Barth, A., 2007].

1.2.3 Pkgtool/Slapt-get

Slackware es una distribución primaria cuyo gestor de paquetes Pkgtool, se conoce como uno de los primeros gestores, lo que hace que las utilidades que posee sean bastante básicas. La herramienta permite las opciones de compilar, instalar, actualizar y desinstalar, es válido destacar que entre éstas no se encuentra la gestión de dependencias. Además brinda la posibilidad de convertir un paquete de formato .rpm hacia un .tgz o .tar.gz para poder instalarlo, con los comandos rpm2tgz y rpm2targz [Slackware Linux Inc., 2005].

También está slapt-get, que es un gestor de paquetes similar al apt-get de Debian, que usa los comandos básicos de gestión de Slackware. Entre las opciones que brinda tenemos que: soporta múltiples fuentes de paquetes, continúa descargas interrumpidas y revisa la integridad de sus paquetes mediante **MD5**. Además posibilita la gestión de dependencias, la búsqueda de paquetes mediante estándares y con soporte de varios protocolos [Wikipedia, 2008e].

Los paquetes gestionados por esta distribución son ficheros compactados que contienen en su interior la estructura de los directorios donde debe quedar copiado cada fichero del software precompilado. En

algunos casos contiene una carpeta *install* con un **script** que se nombra *doinst.sh* que es para opciones post-instalación y que en caso de que el gestor, mientras copia, lo encuentre debe ejecutarlo antes de terminar [Slackware Linux Inc., 2005].

1.2.4 Portage

Es el gestor de paquetes de Gentoo, basado en el sistema de Ports de BSD, utiliza para la gestión de los paquetes la herramienta emerge. Es una herramienta escrita en los lenguajes de programación Python y Bash, ambos son lenguajes interpretados de secuencias de comandos. Esta permite la actualización del árbol de Portage que no es más que una colección de ficheros con extensión .ebuilds estructurados por categorías [Gentoo Linux, 2008a].

Además están las operaciones básicas que se pueden llevar a cabo con el comando *emerge*. Entre ellas la búsqueda de software, instalación, desinstalación, actualización y la gestión de la documentación de los paquetes instalados. Esta herramienta maneja paquetes de código fuente, esta es una ventaja para los desarrolladores permitiendo la configuración personalizada de las aplicaciones. A su vez es la mayor desventaja que presenta este gestor, dado el tiempo que se toma la compilación de cada paquete.

Los paquetes gestionados por este sistema son compactados con los ficheros fuentes, pero se genera por cada uno de ellos un fichero con extensión .ebuild. Este fichero es un script que contiene la información necesaria para descargar, desempaquetar, compilar e instalar los paquetes de código fuente de una aplicación, además de otras opciones pre y post instalación [Gentoo Linux, 2008b].

Los ebuild están escritos en el lenguaje de programación Bash lo cual permite al usuario utilizar los comandos acostumbrados a usar en el shell [Gentoo Linux, 2008a]. Estos paquetes compactados con los ficheros fuentes se van a encontrar en un repositorio. El Portage posee un conjunto de ebuilds y a través de la gestión de estos es que accede a los paquetes en sí. Los scripts ebuild contienen las variables y funciones a usar durante la instalación y la configuración, además de las dependencias a gestionar y las direcciones del paquete.

Los gestores de paquetes son un elemento importante dentro de las distribuciones GNU/Linux, más aún para aquellas que estén destinadas a los usuarios sin conocimiento sobre computación. Por lo que deben

ser capaces de manejar paquetes binarios de forma sencilla facilitando así los procesos de instalación, actualización o eliminación de software para los usuarios. Actualmente las distribuciones cuyos gestores tienen pocas prestaciones con este tipo de paquetes de software trabajan en busca de soluciones.

1.3 Tendencias actuales de metodologías, herramientas y lenguajes de programación para el desarrollo de software libre.

1.3.1 Metodologías de Desarrollo de Software

El proceso de desarrollo de software define actividades prácticas para obtener un producto con calidad disminuyendo los riesgos de fracaso [Jacobson, I.; Booch, G. y Rumbaugh, J., 2000]. Las metodologías de desarrollo son procesos como el expuesto, creadas con distintas características y filosofías de trabajo.

Actualmente hay no pocas metodologías entre ellas se destacan Rational Unified Process (RUP), Microsoft Solution Framework (MSF) y de las Metodologías Ágiles, Xtreme Programing (XP), estas últimas con gran auge en el ambiente del software libre [Zuser, W.; Heil, S. y Grechenig, T., 2008].

El Proceso Unificado o RUP, no es solo una metodología, es un marco de trabajo genérico que pudiera especializarse para diversos tipos de proyectos. Además sigue principios que le son fundamentales como por ejemplo: la planificación del desarrollo por equipo y la documentación del proyecto de forma perdurable y extensible. Esta metodología está definida por tres características fundamentales: es dirigida por casos de uso, centrada en la arquitectura, así como iterativa e incremental [Jacobson, I.; Booch, G. y Rumbaugh, J., 2000].

Los casos de uso representan los requerimientos funcionales que debe cumplir el software, es por ello que sirven de guía para el proceso de desarrollo. No obstante la metodología se centra en la arquitectura pues ésta es una vista del diseño con las características fundamentales del sistema, sin llegar a los detalles. Las iteraciones son el trabajo dividido en partes más pequeñas para que al terminar cada una exista un incremento en el desarrollo del sistema [Jacobson, I.; Booch, G. y Rumbaugh, J., 2000] .

En el caso de MSF es llamado como marco de trabajo (framework) porque al contrario de las metodologías prescriptivas anteriores ofrece flexibilidad y adaptabilidad para varios tipos de software. Esta

metodología basa su filosofía en ocho principios fundamentales: fomentar la comunicación, trabajar hacia una visión compartida, potenciar al equipo, establecer una responsabilidad compartida, enfocarse en la entrega, mantenerse flexible a los cambios, invertir en la calidad y aprender de todas las experiencias. Se puede resumir que éstos se pueden concretar en que no existe un proceso que se aplique óptimamente a los requerimientos de todos los proyectos [Microsoft Corporation, 2003].

Está compuesto por varios modelos que contemplan cada área dentro del desarrollo del software. Algunos de estos modelos son: MSF Team Model (MSF modelo de equipo) y MSF Process Model (MSF modelo del proceso). Además posee tres disciplinas que complementan la correcta aplicación de la metodología que son la de gestión del proyecto, gestión de riesgos y gestión de la preparación [Microsoft Corporation, 2003].

Las metodologías ágiles se caracterizan por un desarrollo rápido y por iteraciones, generalmente incluyen al cliente en el equipo, lo que favorece la calidad y rapidez. Además tienen menos artefactos y roles, es flexible a los cambios durante el desarrollo, menos controlado y con pocos principios [Canós, J. H.; Letelier, P. y Penadés, M. C., 2003]. Algunas de ellas se especializan en determinadas fases del desarrollo, como la programación, la gestión o la modelación. Esto implica que en ocasiones sea necesario combinarlas para llevar a cabo en su totalidad el desarrollo de un proyecto.

Entre las metodologías ágiles existentes se encuentran Scrum, XP, Crystal, Dynamic Systems Development Method (DSDM) y Xbreed. La que más popularidad ha ganado entre los desarrolladores es la metodología Programación Extrema (XP, Xtreming Programing).

La metodología XP según Kent Beck, es ligera, flexible, predecible y para equipos de desarrollo pequeños y medianos. Se basa en cuatro valores fundamentales que son la simplicidad, la comunicación, el reciclado continuo de código o retroalimentación (refactoring) y la valentía. También sigue principios de soluciones rápidas, simples, con cambios incrementales, flexibilidad para adoptar cambios y trabajo con calidad [Beck, K., 1999].

Esta metodología se rige además por doce prácticas entre las que se destacan: el juego de planificación, entregas pequeñas, diseños simples, probar constantemente, retroalimentación, programación en parejas, integración continua y cliente como parte del equipo. Lo que indica que es una metodología orientada al

desarrollo de un software, no realiza la gestión, ni el modelado [Beck, K., 1999].

Scrum es una metodología ágil pensada para la gestión de software, fase muy importante dentro del desarrollo de los mismos. También se caracteriza por su adaptabilidad a varios modelos de calidad, entre ellos el Modelo de Madurez de Capacidad de Integración (CMMI) [Gracia, J., 2006]. Es una metodología ideal para proyectos con requerimientos variables, que requieran de rapidez de desarrollo y flexibilidad [Wikipedia, 2008f]. Se aplica dividiendo el proyecto en iteraciones, llamadas sprint, para las cuales se define el desarrollo de una funcionalidad. Define métodos de gestión y control para complementar la aplicación de otras metodologías ágiles como XP.

Las metodologías ágiles utilizan el Lenguaje Unificado de Modelado (UML) para la modelación, pero no establecen métodos para ello. Sin embargo el Modelado Ágil (AM) indica como modelar un software ha desarrollar con metodologías ágiles. Este no es una metodología en sí misma, sino más bien una filosofía de trabajo para modelar software.

El AM según definición Scott Ambler es una práctica efectiva para el modelado y documentación de software. Sus valores son una extensión de los de XP la simplicidad, la comunicación, el reciclado continuo de código o retroalimentación (refactoring), la valentía y agregan uno nuevo, la humildad. Los principios fundamentales que sigue son: retroalimentación rápida, crear múltiples modelos y a su vez cada uno con un propósito, no documentar modelos innecesariamente, darle mayor importancia al contenido que a la representación, además de los presentes en XP [Ambler, S. W., 2007].

La metodología XP y AM pueden integrase con facilidad, visto que esta última incluye sus mismos valores y principios. Entre las prácticas que propone se encuentran crear varios modelos en paralelo, aplicar los artefactos correctos e iterar hacia otro artefacto. También se incluye la modelación con pequeños incrementos, aplicando patrones y estándares, además de poner estos modelos a disposición de todo el equipo [Ambler, S. W., 2007].

Ambler plantea que AM no exige el uso de herramientas para el modelado de software, a menos que se necesite para dar mayor sentido a la situación. En el desarrollo de esta investigación se utilizará una herramienta de modelado sencilla que permita visualizar los diagramas empleados.

Las herramientas Computer Aided System Engineering (CASE), no son recomendadas en estos casos por su complejidad. Sin embargo en la comunidad de software libre han surgido software que permiten la elaboración de manera sencilla de diagramas UML, entre los que se encuentran ArgoUML, Dia y Gaphor.

La primera de ellas, ArgoUML es una herramienta de **código abierto**, desarrollada sobre **plataforma Java** sobre la cual se soporta su funcionamiento. Permite exportar los diagramas como imágenes, esta disponible en 10 idiomas incluido el español y la ingeniería inversa entre otras características [CollabNet, 2008].

Por su parte el software Dia fue desarrollado con la herramienta GTK+ e inspirada en Microsoft Visio, por lo que trabaja con distintos tipos de diagramas entre los que se incluyen los de UML. Esta herramienta cuenta entre sus principales funcionalidades con el soporte de ficheros **XML**, puede exportar los diagramas en formato de imágenes e imprimirlos [GNOME Project, 2008].

Por último Gaphor, es un entorno sencillo de modelado, desarrollado para plataformas UNIX aunque es multiplataforma. Se caracteriza por ser un plugin extensible, exportar los diagramas a formato de imágenes y de documentos pdf, y los modelos de clases a formato XMI, entre otras [Devja Vu, 2008].

En el desarrollo de esta investigación no se hará uso de estas herramientas pues en ninguno de los casos se incluye la posibilidad de hacer suficientemente detallada la modelación. Por lo cual se recurre a las herramientas CASE entre las que se destacan Visual Paradigm y Rational Rose Enterprise Edition.

Esta tesis se está desarrollando en ambiente de software libre por lo que es recomendable se utilice Visual Paradigm para la modelación con UML. El cual además facilita una mejor organización visual de los diagramas de diseño y tiene soporte para más de 10 lenguajes de programación, incluidos C++ y Python entre ellos [Visual Paradigm, 2008].

1.3.2 Lenguajes de programación para el desarrollo de software

En la actualidad existe una gran variedad de lenguajes de programación utilizados en el desarrollo de software libre. Entre ellos pueden usarse los intérpretes de comandos (shell) de UNIX o **lenguajes de alto nivel**. Entre los primeros el más conocido y usado es Bash que es un lenguaje portátil, además es el

intérprete de comandos predeterminado de Unix. También es una aplicación que sigue las indicaciones de la **IEEE Posix Shell**, ofreciendo un uso con mayor interacción [Free Software Foundation, 2002].

Entre los lenguajes de programación de alto nivel usados para el desarrollo en el ambiente de software libre se encuentran el C++, Java, Python.

El C++ se considera un lenguaje bastante robusto que abarca varios paradigmas incluyendo la programación orientada a objetos. A pesar de brindar la posibilidad de trabajar tanto a bajo nivel como a alto, es uno de los que menos automatismo trae, lo que dificulta su aprendizaje [Wikipedia, 2008g]. Es un lenguaje portable que no se limita a una arquitectura de hardware específica, además de ser multiplataforma, también brinda la posibilidad de crear aplicaciones modulares [Cplusplus, 2008].

El lenguaje Java es también orientado a objeto y basado en las sintaxis de C y C++, haciendo que sea un lenguaje robusto que además mejora las dificultades de sus predecesores. Entre sus principales características cuenta con un modelo de objetos más simple y la eliminación de las herramientas a bajo nivel. Además es multiplataforma y contiene un “recolector de basura” que es el responsable de administrar la memoria solicitada dinámicamente [Wikipedia, 2008h]. Este lenguaje permite crear además de aplicaciones de escritorio, software para la Web y sus servicios. También permite desarrollar sistemas para otros dispositivos electrónicos que no sean solamente computadoras [Java, 2008].

Por otra parte Python es un lenguaje multiparadigma, que incluye el orientado a objetos, que permite la división del programa en módulos reutilizables. Además contiene una gran colección de éstos que permiten realizar varias operaciones. Utiliza tipos de datos dinámicos, la ligadura dinámica de métodos, que no es más que enlazar un método y el nombre de una variable durante la ejecución [Wikipedia, 2008i].

Es un lenguaje que ofrece un fuerte soporte de integración con otros lenguajes y herramientas, permitiendo crear extensiones para sistemas ya existentes [Python, 2008]. Es de destacar que además es un lenguaje interpretado, lo cual significa ahorro considerable en el tiempo de desarrollo del programa.

Para el desarrollo rápido de una herramienta libre que permita la generación de paquetes binarios para un repositorio se propone el uso de Python. El empleo de este lenguaje para la realización de este sistema brindará la posibilidad de reutilizar código, de agregar posteriormente módulos. Además de que el hecho

de ser un lenguaje interpretado favorecerá el rápido desarrollo del sistema.

Las herramientas para el desarrollo con Python son muchas dado que los lenguajes interpretados en su mayoría pueden ser escritos en un editor de textos común. En los sistemas GNU/Linux hay varios editores de texto que tienen la característica de reconocer código de varios lenguajes entre ellos Python. Sin embargo también existen **Ambientes Integrados de Desarrollo** (IDE por sus siglas en inglés) para el desarrollo de sistemas con este lenguaje de programación.

Entre las herramientas más populares para el desarrollo con Python dentro de la comunidad dada sus características son Eric, Boa Constructor y Ulipad. Este último es un editor bastante dinámico y flexible, basado en **wxPython**, que utiliza Mixin y Plugin como técnicas de arquitectura. Este se caracteriza por tener un explorador de las clases así como de los directorios, permite el autocompletamiento de código, puede ser fácilmente extensible y en versiones actuales soporta el trabajo con expresiones regulares [Limodou, 2007].

Por su parte Eric es un IDE escrito en Python para el desarrollo con este lenguaje y con Rubi, distribuido bajo la Licencia Pública General (GPL). Este IDE posee autocompletamiento de código, un depurador de código para Python que soporta la depuración de aplicaciones multihilo. También incluye facilidades avanzadas para la gestión de proyecto y un gestor de tareas. Además tiene integrada una herramienta como plugins llamada Bicycle Repair Man para el refactoring, entre otras características [Eric Python IDE, 2008].

Sin embargo Boa Constructor es un IDE que al igual que Eric es distribuido soportado por la licencia GPL, permite la creación de interfaces de usuarios gráficas (GUI por su siglas en inglés) a través de la herramienta wxPython. Además incluye el completamiento de código, un depurador avanzado, plantillas de código, entre otras características [Wikipedia, 2008j].

1.4 Tendencias actuales en la gestión de paquetes

Algunas de las distribuciones de GNU/Linux que incluyen gestores de paquetes para el trabajo con paquetes de fuentes han comenzado a trabajar en una herramienta que les permita la generación de paquetes binarios para varias arquitecturas. Entre éstas se encuentran Red Hat, Suse y una distribución

nueva denominada Ututo y que tomó como base a Gentoo Linux.

En el caso de la distribución Suse ha desarrollado una plataforma de desarrollo completa y de código abierto que proporciona una infraestructura para el desarrollo de distribuciones basadas en OpenSUSE. A esta plataforma la han denominado como OpenSUSE Build Service y ofrece herramientas de compilación, distribución y publicación de proyectos para diversas arquitecturas [OpenSuse, 2007].

Las herramientas de compilación que contiene permiten la compilación y desarrollo de paquetes para otras distribuciones y para varias arquitecturas. Posee además una interfaz Web que permite la comunicación con ésta de servicios externos (Forge) y sitios Web, permitiéndole usar sus servicios. También tiene la capacidad de crear distribuciones completas o modificar imágenes de éstas [OpenSuse, 2007].

La nueva distribución de Ututo XS incluye un kit de aplicaciones que consiste en un sistema creado para la compilación de los paquetes almacenados en el Portage para las distintas arquitecturas. Conformando con ellos un repositorio de software con todas las aplicaciones precompiladas, es decir, con los paquetes de código binario [Rizzo, P. M., 2007].

Esto favorece el proceso de instalación de aplicaciones en estas distribuciones, en cuanto a tiempo y espacio. Este sistema durante el proceso de compilación y creación de los paquetes gestiona las dependencias. Además busca la versión más actualizada y estable para cada una de ellas, permitiendo con ello un sistema más actualizado [Rizzo, P. M., 2007].

La comunidad de Red Hat en conjunto con la de Fedora han desarrollado un sistema para la creación de paquetes binarios RPM denominado Koji. Es un sistema portable que posee un amplio modelo de datos y una interfaz Web con sistema de autenticación basado en la plataforma Kerberos. Los componentes de éste pueden trabajar en recursos separados mientras puedan mantener la comunicación entre ellos [Affolter, F., 2007].

Para mantener protocolos de seguridad utiliza el **network file system** (NFS) en modo de solo lectura. Koji mantiene una base de datos en la que le da seguimiento a los contenidos del **buildroot**. Además utiliza la **API** de **XML-RPC** para facilitar la integración con otras herramientas [Affolter, F., 2007].

Conclusiones Parciales

Se puede concluir hasta el momento que para el desarrollo de una herramienta de la distribución Nova debe usarse software libre. Por tanto para llevar a cabo la modelación de la solución deben emplearse XP y AM que permiten flexibilidad y rapidez además de incluir al usuario en el desarrollo del software. Para desarrollar la herramienta se deben utilizar los lenguajes interpretados Bash y Python pues poseen mejor interacción con la gestión de los sistemas GNU/Linux. La herramienta a utilizar para modelar y diseñar el software será el Visual Paradigm. Para el desarrollo, dado que no es requisito crear una interface gráfica sino en modo texto, es recomendable el uso de editor más flexible, por lo que se utilizará Ulipad.

Capítulo 2: Kit para la generación de paquetes binarios

Las distribuciones de GNU/Linux más populares entre los usuarios son aquellas que gestionan paquetes binarios debido a las facilidades de trabajo que brindan. La distribución Nova pretende ser un poco más que una personalización de Gentoo. Motivo por el cual se trabaja en buscar una solución para gestionar paquetes binarios con facilidad para todos usuarios.

En el presente capítulo se explica cómo se lleva a cabo actualmente la gestión de paquetes mostrando de esta manera la situación problémica de la investigación. Además se presenta la propuesta de solución y los inicios de su desarrollo a través del uso de la metodología seleccionada.

2.1 Proceso de gestión de paquetes binarios para la distribución Nova

El desarrollo de la distribución Nova se propone la optimización de procesos en función de las necesidades de sus usuarios. Entre las proyecciones planteadas a mediano y largo plazo se encuentran: una aplicación que facilite la instalación de la distro, un gestor de paquetes binarios y una herramienta con interfaz visual para la configuración de la red. Como parte de la investigación asociada al proyecto, se orientó la investigación del proceso de gestión de paquetes binarios, que favorecerá a la automatización de la generación de los mismos.

En este trabajo se dividirá este amplio proceso de gestión en el de creación de los paquetes binarios o compilación de los paquetes fuentes y el de instalación/eliminación/actualización por los usuarios. Enfocándonos en la optimización del primero de ellos pues este último es desarrollado por otro equipo del proyecto.

Actualmente en la distribución Nova el proceso de generación de los paquetes binarios se maneja de forma manual. Se compilan los paquetes fuentes, con el comando *emerge* y las opciones -b o -B, guardándose los paquetes binarios resultantes en el directorio /usr/portage/packages/All.

El uso de las opciones –b y –B indica al comando *emerge* que compile los paquetes de fuentes, sin necesidad de que los instale. En el caso de la opción –b instala el paquete y guarda el binario creado en el

directorio /usr/portage/packages/All. Sin embargo la opción –B solamente crea el paquete binario y lo guarda en la dirección indicada anteriormente. Es importante tener en cuenta que en ambos casos las dependencias son instaladas previamente.

Durante el proceso de generación de los paquetes binarios el usuario que está llevando a cabo el proceso debe gestionar los errores producidos. Cuando el comando *emerge* encuentra un problema durante la ejecución de una tarea, la interrumpe informando del error ocurrido. En ese instante el usuario que atiende el proceso debe intentar solucionar el error y reiniciar el proceso de compilación donde se interrumpió.

Los paquetes binarios obtenidos son ficheros comprimidos que contienen la estructura de directorios de cómo debe quedar el software dentro del sistema de ficheros al ser instalado, entre otros datos de importancia, como el ebuild propio del paquete. Luego se copian para un repositorio los paquetes binarios obtenidos, para que éstos estén disponibles para el resto de los usuarios. Para acceder al repositorio, en el sistema del usuario final debe especificarse en la variable PORTAGE_BINHOST del fichero /etc/make.conf la dirección donde se encuentran los mismos.

Realizado el proceso de generación de los paquetes binarios, los usuarios pueden llevar a cabo los otros procesos contenidos dentro de la gestión de paquetes. De igual manera que el anterior a través del comando *emerge*, no obstante en este caso se utilizarán las opciones -g, -G, -k o -K.

Las opciones -g y –G básicamente son iguales, es decir, se conectan a la dirección guardada en la variable PORTAGE_BINHOST, para descargar los paquetes binarios solicitados y sus dependencias. Las dependencias de un software se refiere a otros sistemas que necesitan estar instalados para un correcto funcionamiento del primero. La diferencia entre estas opciones es que en el caso de –g verifica los paquetes que ya se han descargado para no volver a hacerlo, lo cual no hace –G.

En el caso de –k y –K son opciones para señalar al comando *emerge* que esta manejando paquetes binarios. Existen ciertas diferencias, la opción –k trabaja con los paquetes que están disponibles y –K solo continúa si todos los paquetes binarios están disponibles en el momento en que se analizan las dependencias.

2.2 Propuesta de Solución

Para lograr una optimización de la gestión de los paquetes binarios se propone una herramienta que se encargue de automatizar el proceso de generación de éstos. También se procura que sea capaz de informar los fallos ocurridos o eliminación de la tarea durante la compilación. La automatización de estos procesos a través de una herramienta de software es la solución propuesta por este trabajo.

El Kit se diseñará de manera que permita definir varias opciones a tener en cuenta durante la compilación fundamentalmente las arquitecturas para las que desee llevar a cabo el proceso. Otra opción será la de procesar una lista de paquetes pendientes a generar que almacenará el sistema o establecer una lista con paquetes específicos.

En caso de seleccionar esta última deberá confirmarse el repositorio a utilizar para llevar a cabo la compilación. Además de controlar lo que se compilará, realizará una gestión sobre los errores, informando sobre los mismos y reanudando el proceso.

El desarrollo de la herramienta se realizará de manera modular, conteniendo 2 módulos para organizar el trabajo del sistema. Además se creará uno adicional como módulo principal y será el que enlace a los otros 2. Este diseño modular permitirá que en versiones posteriores se le puedan agregar otras funcionalidades al sistema para ampliar sus opciones. Los módulos serán:

Módulo de Gestión de la Compilación: Es el módulo encargado de llevar a cabo la generación de los paquetes binarios, a través de la compilación de los paquetes fuentes, con la configuración definida y de verificar los errores que pudieran ocurrir.

Módulo de Reportes: Se encarga de generar los reportes sobre el estado en que terminó la compilación, además de reportar las tareas incumplidas por el sistema.

Módulo de Control: Es el módulo encargado de conectar a los otros dos módulos y de guiar el trabajo del sistema.

2.3 Planificación de desarrollo de la Propuesta

La metodología de desarrollo XP posee 4 fases dentro de su ciclo de vida; Planificación, Diseño, Implementación y Pruebas. La primera de ellas es la que se desarrolla en esta sección, como inicio del desarrollo de la herramienta. En esta fase se realizan reuniones con los usuarios de la herramienta para determinar las necesidades principales que esta debe cubrir.

Entre los artefactos más importantes de esta fase se encuentran las historias de usuario, las cuales sirven de guía durante el proceso de desarrollo de esta metodología. Son elaboradas por los propios clientes describiendo los requisitos que deberá cumplir el sistema tal y como ellos lo entienden [Jeffries, R.; Anderson, A y Hendrickson, Ch., 2000]. De esa manera se conforman sus historias a partir de las cuáles se realiza la planificación del proceso de desarrollo con la metodología seleccionada.

Con las historias de usuarios elaboradas se comienzan las reuniones con el equipo de desarrolladores, para elaborar el plan de liberaciones. Esta planificación se basa en las estimaciones de los programadores de cuánto tiempo requerirá la implementación de cada historia [Letelier, P. y Penadés, M. C., 2008]. Además de las necesidades prioritarias, que en base a esto establezcan los usuarios, de ahí la importancia de integrarlos al equipo.

Con el plan de liberaciones se lleva a cabo también la determinación de las tareas de ingeniería que son los grupos de funcionalidades dentro de una misma historia de usuario. Estas serán implementadas por una pareja de desarrolladores en el tiempo que hayan estimado, con lo cual se retroalimenta el plan de liberaciones. Por último el usuario diseñará las pruebas de aceptación que son aquellas que realizará al finalizar una iteración y servirán para determinar si se puede hacer la liberación o no.

2.3.1 Historias de usuario

Las necesidades primordiales de la distribución Nova como usuario principal de la herramienta es la generación de paquetes binarios para su adición a un repositorio. Para esto se establecieron los siguientes requisitos iniciales:

- Establecer una lista de paquetes de fuentes a compilar, incluyendo las dependencias de manera

que se generen los binarios de todos ellos de manera independiente.

- Compilar la lista de paquetes de fuentes para las arquitecturas que se definan.
- Gestionar la ocurrencia de errores de manera que si falla algún paquete no se interrumpa completamente el proceso, solo tener en cuenta la tarea.
- Reportar los errores ocurridos durante el proceso de generación de los paquetes binarios.
- Reportar el estado en que terminó el proceso incluyendo las tareas eliminadas por interrupción del proceso.

Dados los principales requerimientos que debe cumplir la herramienta, fueron conformadas 3 historias de usuario para guiar el desarrollo del sistema:

- La primera contempla el proceso de generación de los paquetes binarios, donde según la lista indicada se compilarán los fuentes. Para ello debe tenerse en cuenta sus dependencias y las arquitecturas de hardware deseadas por el usuario dentro de las que se mostrarán en pantalla. En caso de que no se seleccione ninguna arquitectura de las presentadas se generarán los paquetes binarios para cada una de ellas. La lista mencionada se refiere a un fichero de texto que puede estar en el sistema conteniendo los paquetes que faltan por generar. También puede ser una lista creada por el usuario de los paquetes que necesite generar. (Ver en el Anexo1)
- La segunda implica el proceso de gestión de errores durante el proceso de generación de los paquetes binarios. De éstos deben validarse los más comunes para que el usuario pueda analizarlos al finalizar y darle solución. El sistema generará un reporte (log) sobre lo ocurrido para que pueda ser consultado y analizado posteriormente. Luego debe continuar el proceso a partir de la tarea de compilación que sigue a la fallida, donde se interrumpió, para que el resto de los paquetes sean generados. Los errores o fallos pudieran ser problemas al compilar algún paquete, que puede ser por estar corrupto el paquete o por faltar alguna de las dependencias u otras interrupciones de la compilación. (Ver en el Anexo1)
- La tercera se refiere a que cuando el sistema termine de generar el grupo de paquetes binarios debe generar un informe de estado describiendo el cumplimiento, de las tareas. En caso de la existencia de un error se generará un informe (log) que contenga el estado en que se encontraba el proceso. Además de generar un informe general con todos los fallos ocurridos durante la compilación de un grupo de paquetes. Por estado del proceso de generación de paquetes binarios

se entiende el último paquete compilado correctamente, tiempo, arquitecturas para las que se llevó a cabo la acción, entre otros que pudieran agregarse más adelante. (Ver en el Anexo1)

2.3.2 Planificación de las Historias de Usuario

Para llevar a cabo la elaboración del Release Planing o Plan de Entregas, es necesario hacer una planificación estimada del tiempo que llevará implementar cada historia de usuario. Por el tiempo estimado en esta etapa se entiende el tiempo ideal, que significa trabajar solo en el desarrollo de las historias de usuario sin distracciones. El resultado se debe obtener en aproximadamente entre una y tres semanas, logrando iteraciones pequeñas.

No.	Nombre de HU	Prioridad	Esfuerzo(días)	Iteración
1	Generar paquetes de código binario.	Alta	10	1
2	Gestionar errores.	Alta	7	1
3	Generar informes.	Media	5	2

La estimación elaborada en esta etapa de la planificación permite definir claramente la importancia que tiene cada historia de usuario para el cliente. Con ésto se define la iteración en la que se debe desarrollar la misma para así elaborar el plan de entregas estimado, según el tiempo ideal establecido por los propios desarrolladores.

2.3.3 Plan de Entregas Estimado

En el Plan de Entrega deben quedar claramente definidos los objetivos que serán cumplidos en cada iteración. Además de las historias de usuarios que se desarrollarán en cada una, el tiempo que se demorará y cómo se evaluará la calidad de la misma. Para el desarrollo de la herramienta, queda un plan de entregas (Ver e el Anexo 2) con dos iteraciones que describen:

- Entregar una primera versión de la herramienta con las funcionalidades esenciales por lo que incluye las dos primeras historias de usuario que contienen la generación de los paquetes con la gestión de los errores. La cual será evaluada por el usuario al finalizar, lo cual no debe tardar más de tres semanas para la obtención de una herramienta funcional y con la calidad requerida por el

cliente.

- La segunda iteración pretende refinar el sistema con funcionalidades adicionales que ayuden al trabajo del cliente, hasta el momento solo se incluirá la tercera historia de usuario. Este solo contiene la generación de informes sobre el estado de la compilación y los errores ocurridos durante la misma. Cuando sea terminada la implementación de ésta, lo cual debe significar una semana, será evaluada por el cliente para ser entregada con la calidad requerida.

El plan de entregas elaborado debe ser mostrado al usuario para que de su aprobación y se pueda comenzar con el trabajo de la fase de Diseño. Siguiendo la metodología XP puramente podríamos comenzar dicha fase en este punto, pero debe hacerse un análisis más profundo sobre los requerimientos y las historias de usuario. Este se llevará a cabo a través de la modelación utilizando UML y Modelado Ágil (AM), para mantener la filosofía de agilidad en el desarrollo.

2.4 Modelación de las Historias de Usuario

Las historias de usuarios son el principal artefacto de la metodología XP, describiendo los requerimientos que debe cumplir el sistema. Las mismas incluyen una descripción de las características necesarias, sin embargo para los desarrolladores es importante representarlas en diagramas que permitan modelar el funcionamiento del sistema.

Además ayuda al usuario rectificará si se está modelando la herramienta como esperaba, así cuando se pase a la fase de diseño ocurrirán menos cambios. En este caso haremos uso de un diagrama de flujo para representar el funcionamiento general de la herramienta, para definir con claridad los procesos. Estos últimos se modelarán más específicamente a través de los diagramas de actividades que ayudarán a determinar las tareas de ingeniería.

2.4.1 Diagrama de Flujo (Flow Chart)

Los diagramas de flujo son una técnica estructurada para el desarrollo y para el modelado del negocio, pues se pueden representar los detalles de la lógica del negocio [Ambler, S, 2006a]. Con la ayuda de estos diagramas representaremos de manera gráfica cómo se relacionan las historias de usuario

elaboradas, además del funcionamiento general de la herramienta.

La herramienta primeramente debe permitir configurar cómo se desea que se lleve a cabo la compilación o generación de los paquetes. Durante la compilación deben verificarse la ocurrencia de errores en caso de que ocurran deben reportarse y reanudarse el proceso con el paquete siguiente. Al terminar todo el proceso de generación de los paquetes binarios deben generarse reportes del estado en que quedó la compilación.

Este diagrama evidencia tres procesos fundamentales que coinciden con las historias de usuario expuestas. El proceso de compilación que incluye la definición de la configuración, el de gestión de

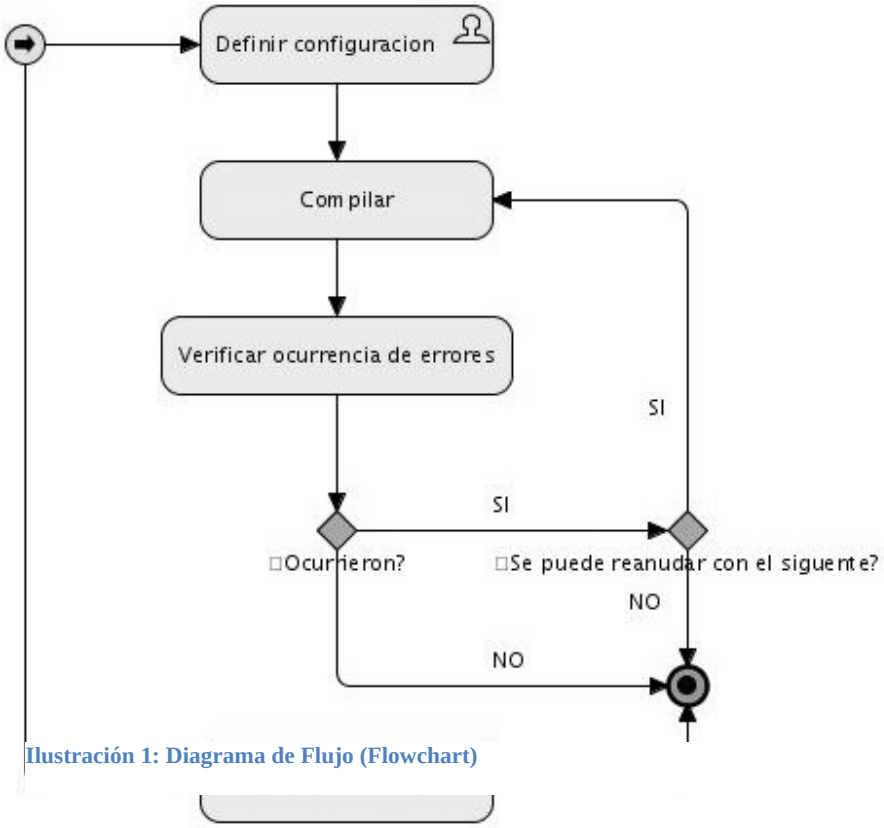

Ilustración 1: Diagrama de Flujo (Flowchart)

errores, que incluye la verificación de ocurrencia de estos y la posible reanudación de la generación de paquetes, y la generación de reportes. Para brindar una mejor representación de éstos se modelará su desarrollo a través de los diagramas de actividades, que son más específicos.

2.4.2 Proceso de Compilación

Este proceso comienza con la definición de la configuración con la cual se quiere sean compilados los paquetes de fuentes. Entre las principales características que se incluyen se encuentra la arquitectura de hardware para la cual se quiere generar los paquetes binarios. En el caso de esta herramienta se brindará

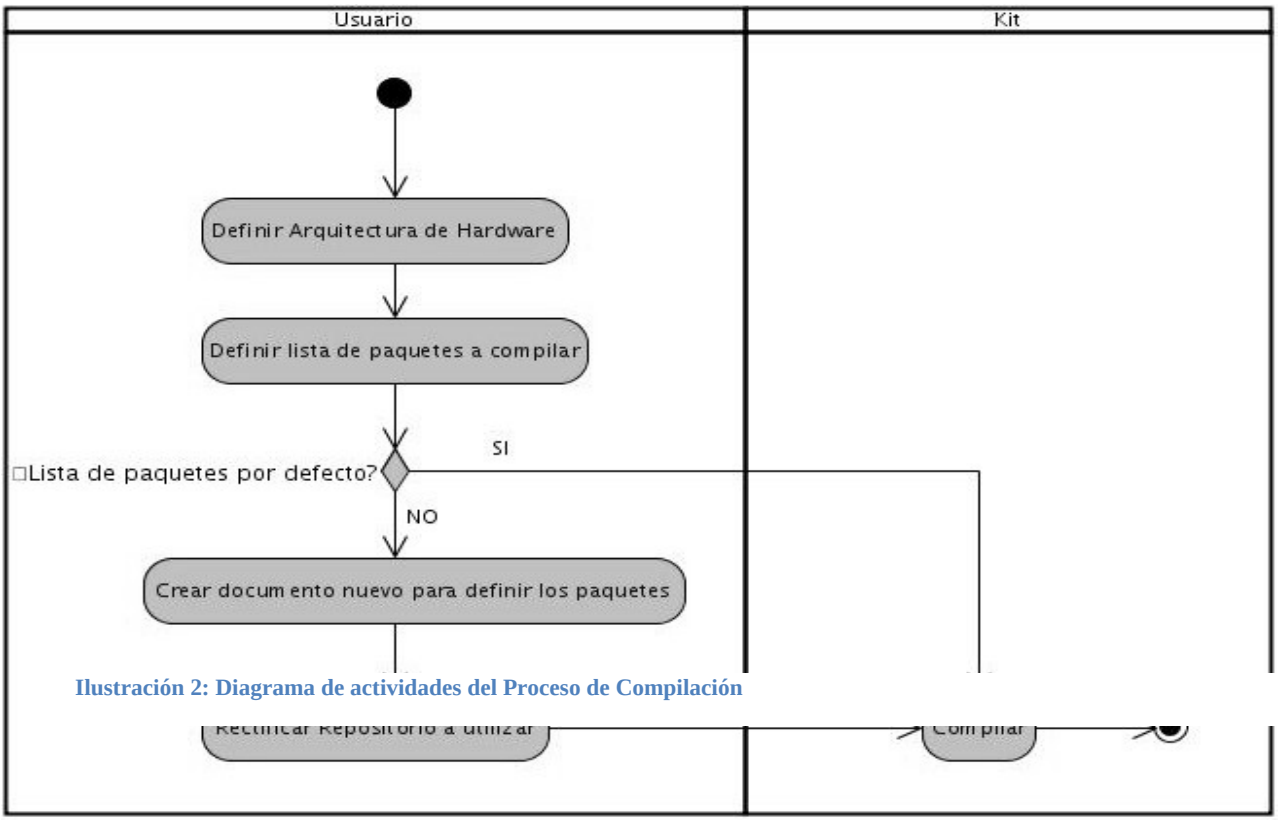

Ilustración 2: Diagrama de actividades del Proceso de Compilación

la opción de compilar para varias arquitecturas en el caso de que no se marque una en específico.

También debe definirse antes de compilar que paquetes se desea que pasen por este proceso, para ello el sistema tendrá una lista con los paquetes que falta por generar. Aunque brindará la posibilidad de crear una lista propia solamente con los paquetes que desee compilar el usuario. En este último caso se debe además rectificar si estos paquetes se encuentran en el repositorio establecido por el sistema o en caso contrario especificar el repositorio de donde se obtendrán los paquetes de fuentes.

2.4.3 Proceso de Gestión de Errores

Durante el proceso de generación de los paquetes binarios pueden ocurrir varios errores. Entre los errores más comunes que se pueden detectar se encuentran: fallo en la conexión al repositorio, no encontrar el paquete en el repositorio, la integridad del paquete corrupta, insuficiente espacio en el disco, paquete marcado con Mask o Testing, entre otros. Para el caso de no encontrar el paquete en el repositorio, el comando *emerge* brinda una lista de repositorios alternativos para la búsqueda del paquete.

Los paquetes marcados con Mask son paquetes recién adicionados y que pueden estar corruptos, con grandes posibilidades de fallar. Los Testing pasaron su etapa Mask, y después de un tiempo prudencial, que puede variar de un paquete a otro, pasan a estar bajo prueba por un mínimo de 1 mes.

La variable KEYWORDS dentro de los ficheros ebuild se usa para definir la arquitectura de hardware con la cual es compatible el paquete. Cuando el paquete está marcado Testing el valor de esa variable tiene delante una tilde (~). Para compilar un paquete en estas condiciones debe actualizarse el fichero /etc/portage/package.keywords, lo que significa agregarle una línea como el ejemplo a continuación, pues por defecto no se aceptan:

Aceptar paquetes en prueba o inestables siempre: *media-libs/libgd ~x86* o *net-analyzer/netcat*

En el caso de los paquetes marcados como Testing o Mask no se le dará solución, dado que en caso de que se compile un paquete en estas condiciones el binario obtenido no sería un paquete estable.

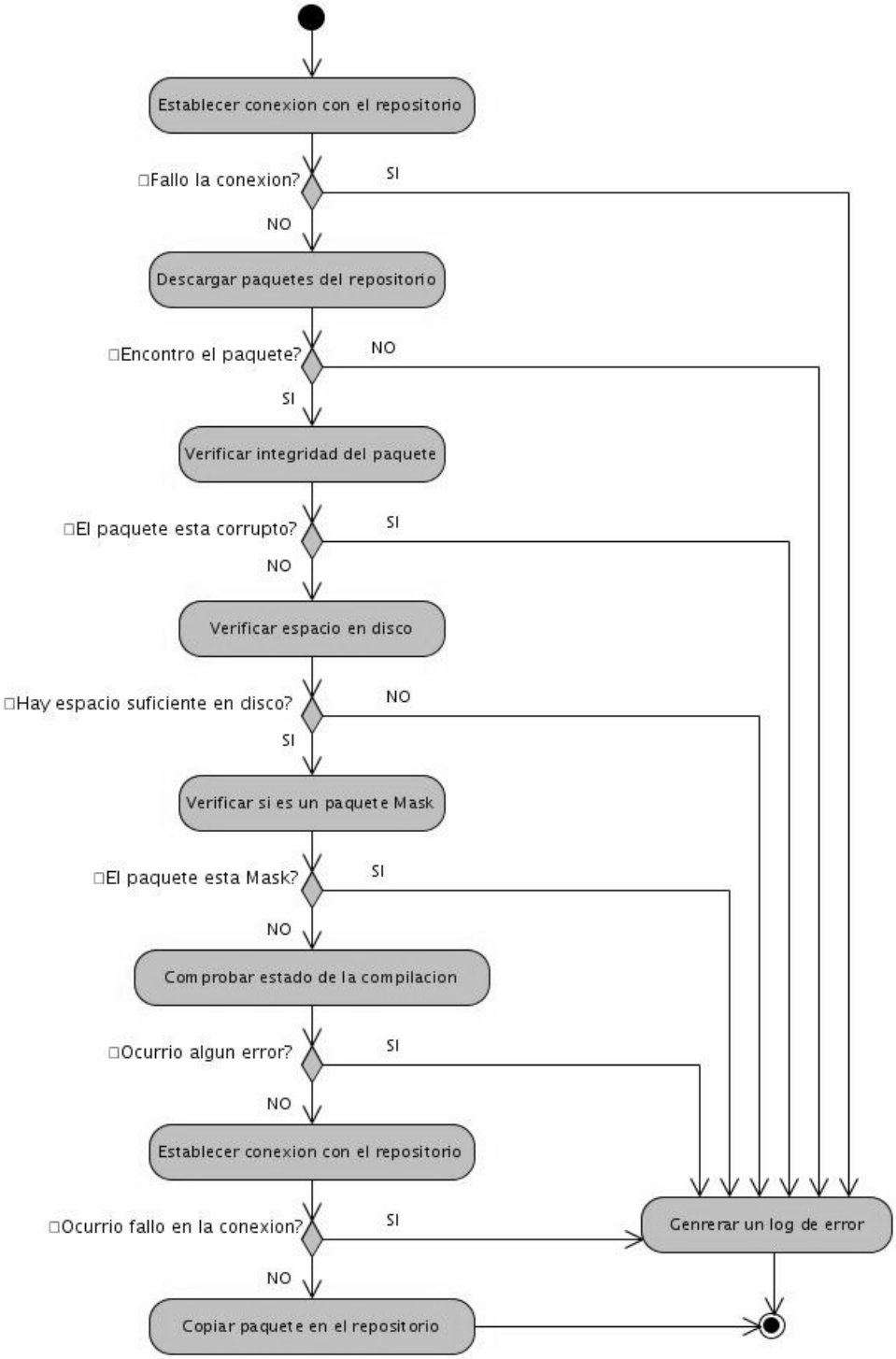

Ilustración 3: Diagrama de actividades del Proceso de Gestión de Errores

En el diagrama de actividades de este proceso se evidencia que no hay interaccion con el usuario, dado

que son acciones realizadas internamente por el sistema. También se aprecia como por cada error se generará un fichero de log, un documento de texto con la bitácora de información (ver en el Glosario) del error ocurrido.

2.4.4 Proceso de Generación de Reportes

Los usuarios entre los requerimientos presentados incluyen la generación de reportes tanto de estado como de los errores ocurridos. Este proceso es similar en ambos casos, puesto que a medida que se realice la compilación se irán generando registros (logs) sobre lo que va ocurriendo. Los reportes en ambos casos serán una compilación de estos a partir de la fecha del último reporte generado.

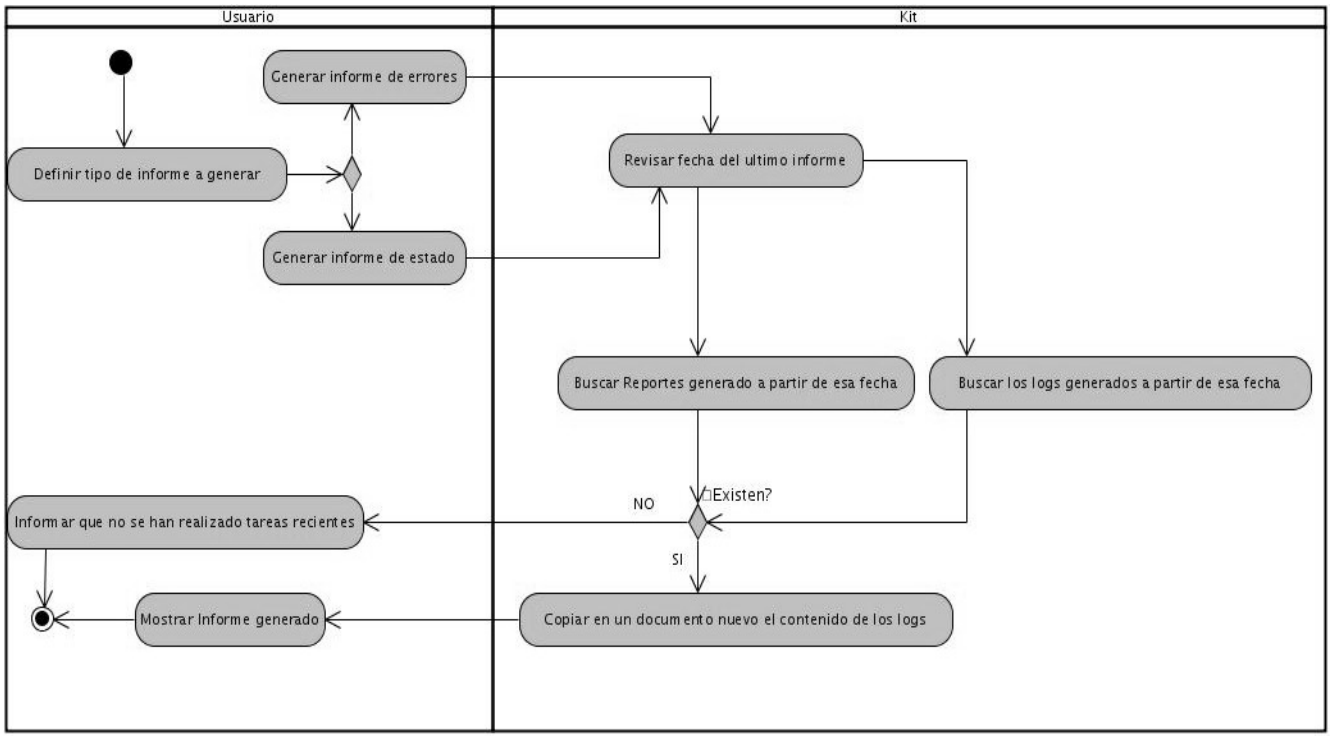

Ilustración 4: Diagrama de actividades del Proceso de Generación de Reportes

2.5 Planificación por iteraciones

El desarrollo de las iteraciones comienza desde la fase de planificación, con la elaboración de las tareas de la ingeniería y las pruebas de aceptación. Las tareas de ingeniería son funcionalidades del sistema que se irán desarrollando por separado y luego serán integradas. Estas se elaboran a partir de las historias de usuario y también requieren de una planificación. Deben ser más descriptivas y sencillas de manera que su desarrollo no tome más de 3 días [Beck, K., 1999].

Las pruebas de aceptación son las encargadas de probar al final de la iteración que funcionen correctamente los requisitos definidos en la misma. Estas deben ser diseñadas por el cliente cuidando que verifiquen la mayor cantidad de funcionalidades incluidas en la historia de usuario para la mayor cantidad de variantes posibles [Gutiérrez, J. J.; Escalona, M. J.; Mejías, M. y J. Torres, 2008]. Luego al final de cada iteración el cliente llevará a cabo las pruebas y dará o no su aprobación sobre la calidad del sistema.

2.5.1 Iteración 1

La primera iteración debe contemplar las funcionalidades principales del sistema, para poder entregar una primera versión funcional para el usuario. En este caso el cliente priorizó las historias de usuario 1 y 2 que son las que contienen la mayor importancia dentro de la herramienta.

La historia de usuario 1 es la principal dentro del sistema pues representa el objetivo funcional fundamental que es la compilación de paquetes de fuentes para generar paquetes binarios. Esta se dividirá en tres tareas de ingeniería esenciales (Ver en el Anexo 3), como se comentan a continuación:

- La primera será la responsable del diseño de la interfaz para la compilación, la cual debe elaborarse de forma que sea cómoda para que el usuario pueda seleccionar la opción a la que desee acceder. Además deben diseñarse las interfaces que mostrarán las distintas opciones de la configuración, en caso de que el usuario determine usar esa opción.
- La segunda de las tareas se encargará de captar la configuración introducida por el usuario y modificar la establecida por el sistema en caso necesario. Para lo cual debe mostrarse las opciones de configuración para la compilación, como son las arquitecturas de hardware para las

cuales se generarán los paquetes, la lista de éstos a usar durante el proceso. Además debe cargarse el fichero de configuración de manera que pueda ser utilizado cómodamente durante la compilación.

- La tercera tarea implementará como se guiará el proceso de compilación a partir de la lista de los paquetes de fuentes definida en la configuración se obtendrá la lista de las dependencias, estas deben ordenarse de manera que puedan compilarse independiente, sin dependencias.

En el caso de la historia de usuario 2 estará compuesta por tres tareas de ingeniería (Ver en el Anexo 3), pues gestiona los posibles errores que se presenten durante el proceso de compilación, estas se describen brevemente como sigue:

- La primera de las tareas contemplará los errores de descarga de los paquetes de fuentes del repositorio. Para ello debe conectarse al repositorio y buscar los paquetes de fuente que se desea compilar. En caso de que los encuentre debe verificarse si hubo alguna falla por espacio insuficiente en disco o por estar el archivo corrupto antes de descargar los paquetes.
- La segunda tarea recopila los errores ocurrido durante la compilación propiamente, chequeando los paquetes marcados como Mask o Testing y registrándolos conjunto con cualquier otro error ocurrido durante el proceso.
- La tercera tarea se encarga de los errores en la copia de los paquetes en el servidor desde el cual estarán disponibles para los usuarios. Para ello debe establecerse una conexión con el servidor para llevar a cabo la autenticación, si no falla se copiará el paquete en la categoría correspondiente.

Las pruebas de aceptación diseñadas por el cliente para verificar las funcionalidades implementadas en esta iteración se concentran en la primera historia de usuario. Esto se debe a que en el caso de la segunda iteración no hay interacción del sistema con el usuario. La verificación de lo implementado en la segunda historia de usuario se verá reflejado en los paquetes generados y copiados en el repositorio.

En esta iteración las pruebas de aceptación se centrarán en la definición de la configuración para la compilación. Se chequeará cada acción implicada con acciones validas e incorrectas, para asegurar la calidad de la herramienta.

Se diseñaron nueve pruebas de aceptación (Ver en el Anexo 4) en las cuales se analizan las diferentes opciones posibles a seleccionar y que pudieran generar un error de funcionamiento de la aplicación:

- El primer caso de prueba chequeará parte de la definición de las arquitecturas de hardware para las cuales se llevará a cabo el proceso de compilación. En este caso no se seleccionará ninguna arquitectura para de esa manera verificar que se generan los paquetes para todas las contempladas por el sistema, como opción predeterminada.
- El segundo caso de prueba se encargará también de la definición de la arquitectura de hardware a utilizar. En esta ocasión el usuario deberá seleccionar varias arquitecturas de las mostradas, de manera aleatoria, verificando que se compilen los paquetes solo para estas.
- El tercer caso de prueba comprobará parte de la selección de la lista de los paquetes a compilar. En esta primera se verificará que la opción de utilizar la lista de paquetes que contiene el sistema genere los binarios, de los paquetes que ésta incluye, correctamente.
- El cuarto caso de prueba es otro de los encargados del chequeo de las opciones de definición de la lista de paquetes. En este se seleccionará la opción de crear la lista, la cual debe mostrar un editor de texto, de fácil manejo, que permita al usuario introducir el nombre de los paquetes que desea generar.
- El quinto caso de prueba trata más directamente la opción de crear la lista de paquetes por el usuario. Este se enfoca en la introducción de nombres de paquetes no válidos a lo que el sistema debe responder lanzando un error al guardar el documento de la lista.
- El sexto caso de prueba se enfoca en la introducción de los nombres de los paquetes a compilar correctamente. Verificando que el sistema al comprobar el fichero que contiene la lista pase al siguiente paso.
- El séptimo caso de prueba está dirigido a comprobar el paso de rectificar el repositorio a utilizar para el proceso de compilación. En este se introducirá un repositorio nuevo, pero con la dirección escrita incorrectamente, lo cual debe reportarse como un error.
- El octavo caso de prueba es otro de los que chequeará la rectificación del repositorio a utilizar. En esta ocasión se escribirá correctamente la dirección del nuevo repositorio, con lo cual debe comenzar la compilación.
- El noveno caso de prueba comprobará que al rectificar el repositorio establecido por el sistema

como el que se desea se utilice para el proceso de compilación, se pase a iniciar el mismo sin problemas.

2.5.2 Iteración 2

La segunda iteración comprende solamente la historia de usuario 3 que describe la generación de informes sobre el estado de la compilación y los errores ocurridos. Esta constará de dos tareas de ingeniería (Ver en el Anexo 3) cada una encargada de la generación de un tipo de informe, ambas procederán de manera similar pero con información diferente, como se comenta a continuación:

- La primera de las tareas es la encargada de la generación de los informes del estado del proceso de compilación. Para ello debe chequearse la fecha del último generado, para comenzar a buscar los logs a partir de esa fecha. En caso de que se encuentren se creará un documento donde se copiarán los datos brindados por los logs, quedando así conformado el informe.
- En el caso de la segunda tarea que generará los informes sobre los fallos ocurridos durante el proceso de compilación. Primero se chequeará la fecha del último informe generado para buscar los logs de errores a partir de esa fecha, si existen estos logs se copiará su contenido en un documento nuevo.

En esta iteración la interacción con el cliente es poca y no está expuesta a errores, aún así se elaboraron pruebas de aceptación para la historia de usuario 3. Las pruebas en este caso verificarán que no se generen informes dobles o incorrectos, por ello solo se elaboraron 2 pruebas de aceptación (Ver en el Anexo 4):

- El primer caso de prueba trata sobre la opción de generar informes sobre el estado del proceso de generación de los paquetes binarios. En la cual el sistema debe mostrar un informe con todo lo ocurrido durante el proceso de compilación. Debe tenerse en cuenta la realización de esta prueba dos veces para chequear que no se generen informes repetidos.
- El segundo caso de prueba comprobará de manera similar al anterior la generación de los informes sobre los fallos ocurridos durante el proceso de generación de los paquetes binarios. Opción con la cual debe mostrarse un informe con los errores ocurridos durante la compilación y por tanto de la

tarea que se canceló. Este caso también debe realizarse en dos ocasiones continuas para verificar que la creación por parte del sistema de informes dobles.

Conclusiones Parciales

Analizando la fase de planificación para la herramienta se puede concluir que la misma se debe poder implementar en un tiempo ideal de 4 semanas. El sistema se entregará con la aprobación por el cliente de la calidad del mismo a través de las pruebas de aceptación elaboradas. Además de las pruebas que deberá llevar a cabo por los desarrolladores del sistema antes de integrar las funcionalidades desarrolladas a la herramienta.

Capítulo 3: Diseño del Sistema

En el ciclo de vida de la metodología XP luego de la fase de Planificación lleva a cabo el Diseño del sistema durante la fase que lleva el mismo nombre. El proceso a partir de este punto define la arquitectura del sistema, de la cual según plantea Kent Beck, una parte es evidenciada por “la metáfora”. Esta no es suficiente por tanto, con las historias de usuario de la primera iteración se traza la estructura base del sistema (“esqueleto”) que permite describir una arquitectura básica, que luego se podrá ir refinando [Beck, K., 1999].

Además se diseñan las tarjetas CRC (Class Responsibility Collaborator), que describen las responsabilidades de las clases y las colaboraciones. El diseño de las clases, es un apartado importante dentro de esta fase, puesto que describe las funcionalidades a implementar en cada una y las variables que contendrán. En este capítulo se expone el diseño desarrollado para la herramienta y la modelación del mismo.

3.1 Arquitectura del Sistema

La arquitectura de software ha sido conceptualizada por varios autores desde distintos puntos de vista, entre los más renombrados se encuentran la recomendación de la IEEE 1471/2000, Paul Clements y la compilación del SEI (Software Engineering Institute). El primero de ellos refiere: “La arquitectura de software es la organización fundamental de un sistema encarnada en sus componentes, las relaciones entre ellos y el ambiente y los principios que orientan su diseño y evolución” [Reynoso, B., 2004].

Clements por su parte la define como: “una vista del sistema que incluye los componentes principales del mismo, la conducta de esos componentes según se la percibe desde el resto del sistema y las formas en que los componentes interactúan y se coordinan para alcanzar la misión del sistema” [Reynoso, B., 2004].

Por último la compilación del SEI la expone como: (1) el trabajo dinámico de estipulación de la arquitectura dentro del proceso de ingeniería o el diseño, (2) la configuración o topología estática de sistemas de software contemplada desde un elevado nivel de abstracción y (3) la caracterización de la disciplina que se ocupa de uno de esos dos asuntos, o de ambos [Reynoso, B., 2004]. Estos tres conceptos podrían

resumirse en que la arquitectura de software organiza los componentes del sistema y sus interacciones de manera que permita visualizar la estructura del mismo.

En la metodología XP según plantea Kent Beck la arquitectura se obtiene a través de “la metáfora” y además se tomarán las historias de usuario de la primera iteración para crear la estructura esencial del sistema [Beck, K., 1999].

3.1.1 Metáfora

La metáfora descrita por Beck es una historia que permite a todos en el equipo, incluido el cliente, explicar con las mismas palabras como funcionará el sistema. Ward Cunningham y Martin Fowler la describen como un sistema de nombres que actúa como un vocabulario para hablar del dominio del sistema. Seleccionada correctamente la metáfora ayuda con la modelación y diseño del sistema, además de que permite comprender mejor el dominio [Fowler, M., 2004].

La metáfora del sistema propuesto sería: Herramienta que gestiona la generación de paquetes binarios, con una configuración dada por el usuario, manejo de errores y creación de informes. Se puede ver representada a través del diagrama de flujo y los diagramas de actividades que describen los procesos involucrados.

3.1.2 Arquitectura

Para definir la arquitectura es necesario determinar los elementos fundamentales del sistema, las clases, módulos u objetos principales y sus relaciones. En el caso de XP, Kent Beck sugiere que se diseñe una arquitectura con los elementos de la primera iteración que servirá como base y luego se refinará durante las otras iteraciones [Beck, K., 1999].

Scott Ambler plantea como modelar la arquitectura de manera ágil sin obviar elementos que no estén presentes en la primera iteración. Para ello propone el uso de varias vistas, referenciando a otros autores que lo sugieren así, dado que permite tener una mejor visión del sistema. Para las vistas se utilizarían los diagramas de navegación, que son aquellos que ayudan a representar la organización del software [Ambler, S., 2008].

Para mantener la filosofía ágil es importante el desarrollo solo de los diagramas necesarios dando seguimiento así al principio de "viajar ligero", logrando un diseño simple. Para la descripción de la arquitectura de la herramienta propuesta la autora de la investigación propone realizar un diagrama de componentes [Ambler, S., 2008].

Este tipo de diagramas es utilizado para mostrar la organización y las dependencias lógicas entre un conjunto de componentes de software que pueden ser ficheros de código fuente, librerías, binarios o ejecutables. Se tienen en cuenta los requerimientos de facilidad de desarrollo, la gestión de software, la reutilización y las restricciones impuestas por los lenguajes de programación determinados a utilizar durante el desarrollo.

Para el desarrollo de esta investigación se elaboraron dos diagramas de componentes que mostrarán la estructura del software desde dos puntos de vista. Uno muestra los módulos y sus relaciones, el segundo representa como componentes la unidad más simple dentro del sistema las librerías con las clases.

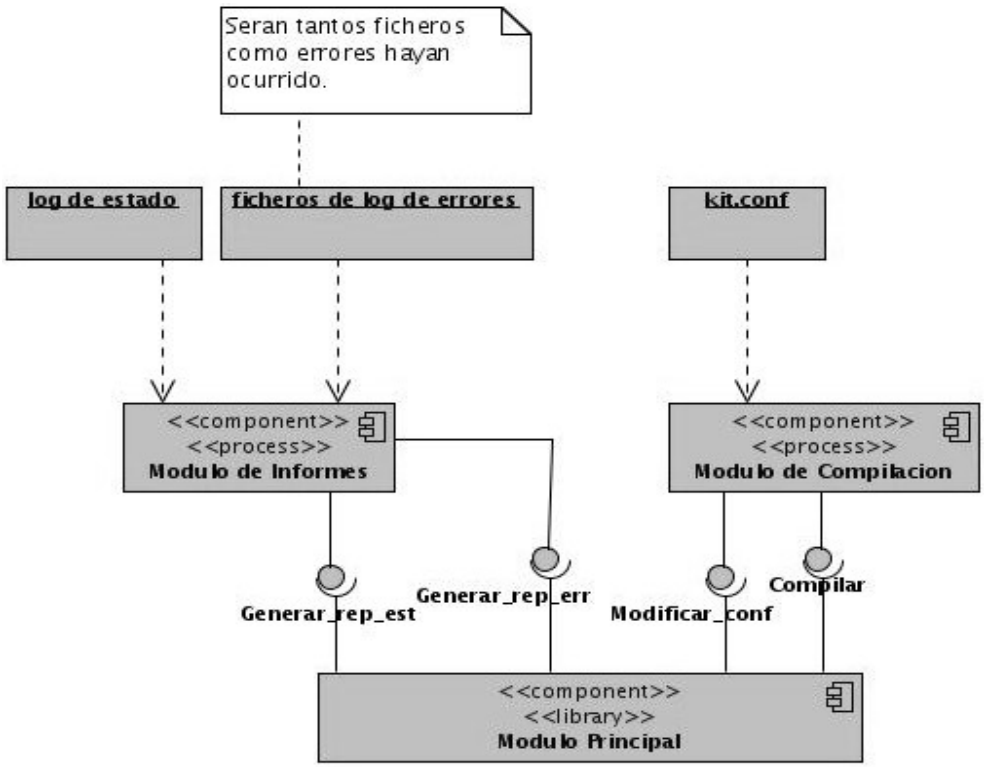

Ilustración 5: Diagrama de componentes por módulos

En el diagrama se puede apreciar los módulos que componen el sistema relacionados entre sí a través de las interfaces. Estas últimas en este caso no se refieren necesariamente a una interfaz visual como generalmente se le conoce. Se muestra cómo el módulo de compilación depende del fichero de configuración y se comunica con el módulo principal mediante las funcionalidades que permiten modificar dicho fichero y la de compilar.

En el caso del módulo de reportes depende de un grupo de ficheros de log de errores y de los log de estado que se generarán cuando se lleve a cabo la compilación. Este se comunica con el módulo principal a través de los procedimientos de generar reportes de estado o de errores.

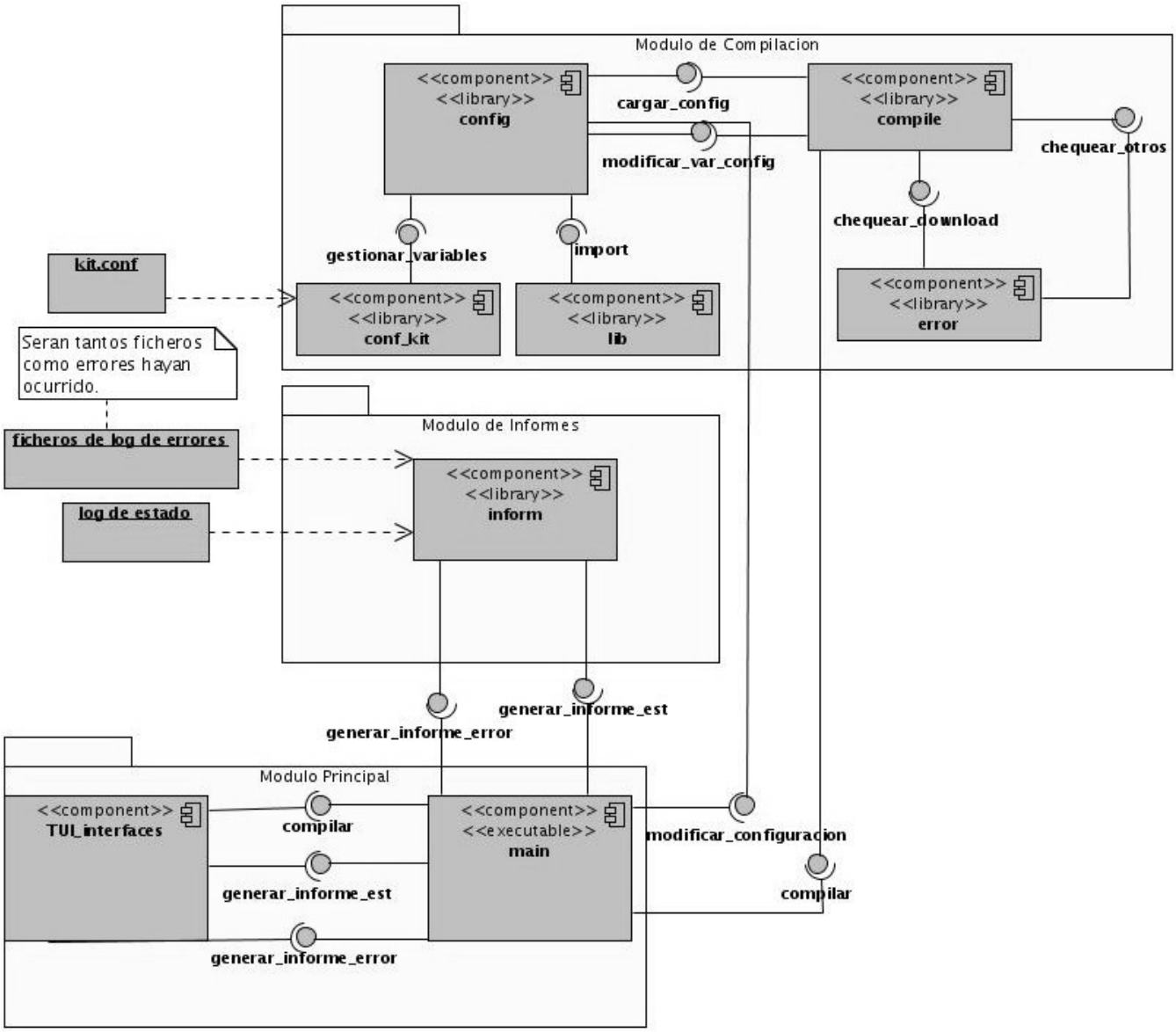

Ilustración 6: Diagrama de componentes detallado por ficheros

En este caso los componentes representan cada fichero del sistema, y pueden contener código Python o datos a utilizar por el sistema. De esta manera se aprecia con un nivel de detalle mayor la estructura de la herramienta que se deberá tener en cuenta durante su desarrollo. En este diagrama se agruparon los componentes por paquetes que representan los módulos mostrados anteriormente.

3.2 Tarjetas CRC

Las tarjetas CRC son otra etapa dentro de la fase del diseño, que permite a los desarrolladores enfocarse hacia el diseño de las clases dentro del sistema. Son una técnica de modelado introducida inicialmente con el objetivo de enseñar los conceptos del paradigma orientado a objeto. Están divididas en tres secciones que representan los objetos de las clases, sus responsabilidades y las clases que colaboran con el [Ambler, S., 2006b].

Las clases representan un conjunto de objetos, en las tarjetas CRC se escriben en singular dado que representan a una instancia en particular. Las responsabilidades son los datos que el objeto conoce o las acciones que es capaz de realizar, para las cuales en ocasiones no tiene la información necesaria. Las colaboraciones son las clases que le pueden brindar los datos necesarios para completar sus responsabilidades [Ambler, S., 2006b].

La metodología XP plantea el uso de las tarjetas CRC como parte de sus reglas o prácticas para el diseño por los detalles que aportan al diseño de las clases dentro del sistema. Este proceso incluye a todo el equipo de trabajo lo cual le aporta al diseño mayor cantidad de ideas para así obtener un resultado mejor.

La herramienta propuesta consta de 4 clases fundamentales, una encargada de gestionar la configuración almacenada en un fichero, la responsable de la compilación de los paquetes de fuentes. Otra que manipulará los errores que deben ser chequeados en el sistema y la que generará los informes, por lo que el modelo CRC quedaría de la siguiente manera:

Clase: Configuración (config)	
Responsabilidad: ➢ Cargar fichero de configuración. ➢ Modificar configuración.	Colaboradores:
Clase: Compilación (compile)	

Responsabilidad:	Colaboradores:
➢ Obtener la lista de todos los paquetes a compilar, incluidas las dependencias. ➢ Ordenar la lista de los paquetes a compilar. ➢ Compilar lista de paquetes.	➢ Configuración ➢ Errores

Clase: Errores (error)	
Responsabilidad:	Colaboradores:
➢ Chequear descarga de paquetes de fuentes. ➢ Chequear otros errores en la compilación. ➢ Chequear copia hacia el repositorio de paquetes binarios.	➢ Compilación ➢ Configuración

Clase: Informes (inform)	
Responsabilidad:	Colaboradores:
➢ Generar informe de errores. ➢ Generar informe de estado.	

Las tarjetas CRC permiten hacer un diseño más acertado del sistema desde el punto de vista de los desarrolladores, dado que representan las clases que va a contener el sistema.

3.3 Diseño de Clases

Las clases representan al igual que en las tarjetas CRC un conjunto de objetos con características similares, siendo uno de los principales componentes de una aplicación orientada a objetos. Los diagramas de clases en la fase del diseño son de gran utilidad para los desarrolladores, puesto que representa las clases, sus atributos, métodos y relaciones.

Las clases diseñadas deben ser consistentes con la arquitectura, dado que estas clases son diseñadas de manera muy similar a como serán implementadas. El lenguaje utilizado para diseñar estas clases esta en correspondencia con el lenguaje a emplear por los desarrolladores para permitir un mejor entendimiento. Las relaciones entre las clases son las mismas empleadas en la programación: generalización (herencia) y asociación (agregación/composición).

3.3.1 Clases del diseño

Las clases en el diseño presentan cómo debe ser construido el sistema, según el lenguaje de programación y en caso de que se use, el estándar de programación a aplicar. La herramienta propuesta cuenta con 4 clases en su diseño, tres de ellas pertenecen al módulo de compilación y el de generación de informes que siguiendo los requisitos solo necesita una clase.

Nombre: config	
Tipo de clase: manejadora de datos	
Atributo	Tipo
Para cada responsabilidad:	
Nombre:	get_config
Descripción:	Carga el contenido del fichero de configuración de manera que se pueda obtener los datos necesarios durante la compilación.
Nombre:	change_config
Descripción:	Modificar el valor de una variable en el fichero de configuración.

Nombre: compile	
Tipo de clase: controladora	
Atributo	Tipo
Para cada responsabilidad:	
Nombre:	get_list
Descripción:	Dada la lista de paquetes que se pretende compilar, buscar los paquetes disponibles con sus dependencias, creando una lista de los que se compilaran.
Nombre:	sort_list
Descripción:	Ordenar la lista de paquetes que se compilarán para evitar errores por dependencias durante la compilación.
Nombre:	compile_list
Descripción:	Ordenarla compilación teniendo en cuenta los posibles errores durante el proceso.

Nombre: error	
Tipo de clase: controladora	
Atributo	Tipo
Para cada responsabilidad:	
Nombre:	check_download
Descripción:	Chequea que el sistema se conecte al repositorio, encuentre el paquete, y la computadora tenga espacio en disco para copiarlo.
Nombre:	check_others
Descripción:	Chequea la ocurrencia de errores durante la compilación.
Nombre:	check_upload
Descripción:	Chequea que el sistema se autentique y conecte al repositorio, y este tenga espacio en disco para copiar el nuevo paquete.

Nombre: inform		
Tipo de clase: manejadora de datos		
Atributo		Tipo
Para cada responsabilidad:		
Nombre:	inform_error	
Descripción:	Crear un informe de los errores ocurridos durante la compilación y registrados en ficheros logs.	
Nombre:	inform_est	
Descripción:	Crear un informe del estado de la última compilación registrado en ficheros logs.	

3.3.2 Diagrama de clases del diseño

Las clases fueron diseñadas aplicando el estándar Python PEP 8 (Ver en el Anexo 5), donde lo fundamental es mantener la coherencia y mantener la legibilidad del código. Se determinó el uso de éste, puesto que nuestro sistema se está diseñando en el marco de un proyecto mayor que es la distribución Nova, en la cual se aplica este estándar para los sistemas desarrollados en Python.

Esta directriz de codificación para desarrolladores de Python plantea varias reglas sobre diseño del código, las importaciones de librerías, los espacios en blancos en las expresiones y declaraciones, los comentarios, la documentación, los convenios de asignación de nombres y recomendaciones para la programación.

En el diseño solo se aplicarán las restricciones referentes a la asignación de nombres, puesto que son las que se evidencian en el diagrama de clases del diseño. Estas plantean que los nombres no deben ser largos, pero si descriptivos, en el caso de las clases deben comenzar con mayúscula. En las funciones deben escribirse todo en minúsculas o en mayúsculas y las palabras separadas por subraya (_) [Van

Rossum, G. y Warsaw B., 2001].

Las clases diseñadas para el desarrollo de la herramienta propuesta están relacionadas de la siguiente manera:

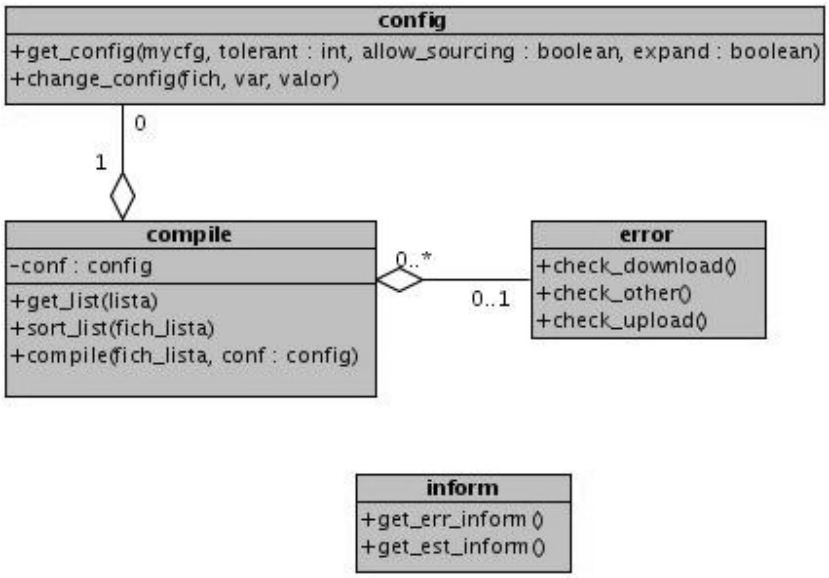

Ilustración 7: Diagrama de clases del diseño

Se puede apreciar cómo la clase compile contendrá un objeto de la clase config que usará para verificar las opciones definidas por el usuario para la compilación. Además podrá contener varios objetos de tipo error que serán los que le permitirán hacer llamadas a los métodos de esta clase para chequear posibles problemas. En el caso de la clase inform no se relaciona puesto que tomará sus datos directamente desde los ficheros de logs generados durante el proceso de compilación.

3.4 Interfaces de Usuario

Las interfaces de usuario son otro elemento importante en la fase de diseño, dado que le dan una visión general al usuario sobre el sistema. La herramienta propuesta cuenta entre sus requisitos el diseño de una interfaz en modo texto cómoda y sencilla. Los lenguajes de programación determinados para el desarrollo del sistema permiten en ambos casos la creación de interfaces con estas características a través de determinadas funciones.

Se sugiere crear una herramienta de manejo sencillo, que brinde las opciones a través de menús donde el usuario seleccione la acción que desee realizar. Implicaría una interfaz por cada uno, en algunos de los cuáles se podrán escoger varias opciones, lo cual cuenta como un requisito adicional para el diseño.

Ilustración 8: Interfaz del menú principal del sistema

Este es el menú principal, que mostrará todas las opciones que brindará el sistema, entre las que están los requisitos pedidos por el cliente y además la opción de mostrar los Datos del sistema. En caso de que seleccione ésta última va a mostrarle al cliente la arquitectura de hardware de la computadora en la que está trabajando.

Ilustración 9: Primer menú de la configuración

En caso de que seleccione la segunda opción se le pedirá que confirme la configuración para la compilación. En primer lugar se le permite determinar las arquitecturas para las que desea generar los binarios, como se muestra en la figura 9. Si no desea seguir puede volver “Atrás” al menú principal o “Salir”, en caso de que continúe se dará la posibilidad de definir que lista de paquetes a compilar, como se muestra en la figura 10.

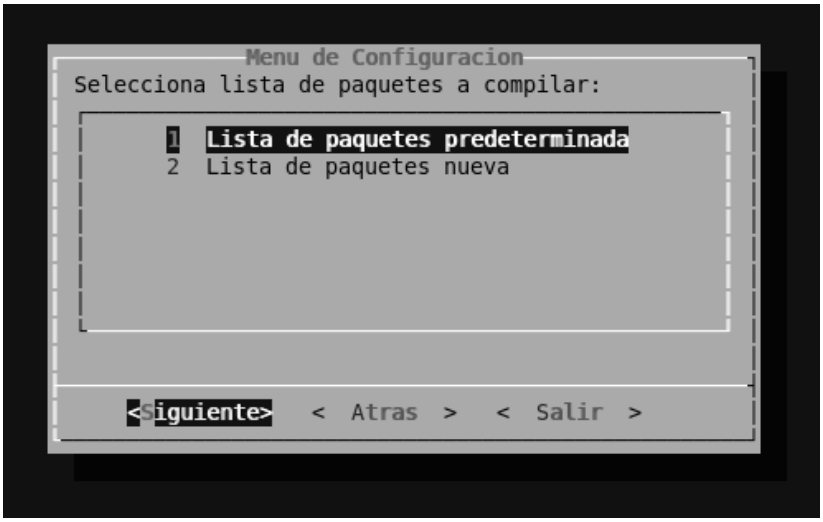

Ilustración 10: Menú de selección de la lista de paquetes

Se puede apreciar como el sistema brinda la opción de crear una lista nueva, lo cual se hará a través del editor de texto “nano” que se muestra en la figura 11. En caso de que necesite compilar la lista predeterminada deberá después confirmar el uso del repositorio establecido de manera predeterminada.

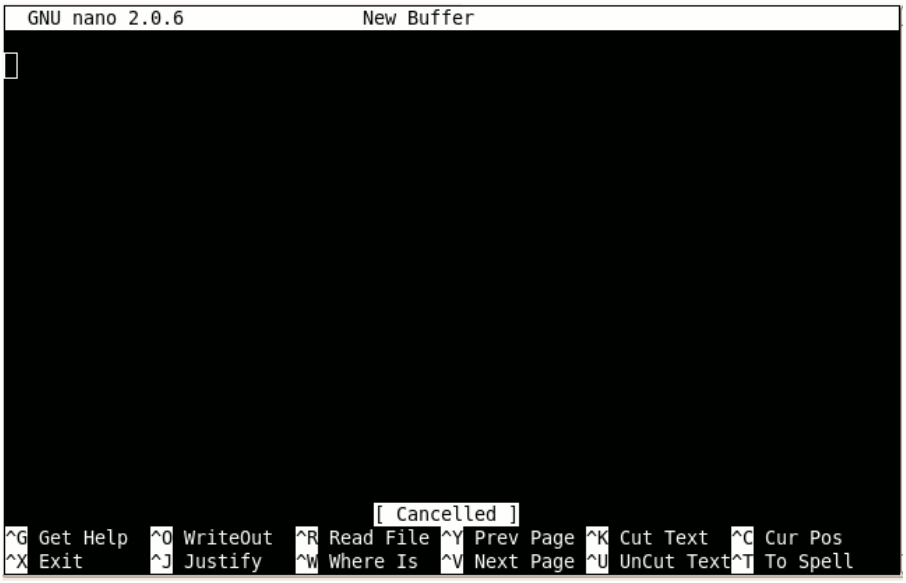

Ilustración 11: Editor de texto para la creación de una lista de paquetes nueva

Para la confirmación del repositorio a utilizar se mostrará un mensaje con el establecido como se muestra en la figura 12 y en caso de que necesite cambiarlo existe una opción para hacerlo.

Ilustración 12: Mensaje que muestra el repositorio a utilizar

En este caso el sistema mostrará una caja de texto, como se muestra en la figura 13, para que el usuario entre la nueva dirección y proceda a la compilación de la lista definida anteriormente.

Ilustración 13: Ventana para modificar repositorio

A través de las interfaces mostradas se confirma o modifica la configuración que utilizará la herramienta para la generación de los paquetes binarios. Las opciones de generar reportes sobre el estado o los errores ocurridos durante ese proceso solo mostrarán en pantalla el fichero del informe creado.

Conclusiones

- El estudio sobre la gestión de paquetes en las distribuciones GNU/Linux permitió apreciar la mayor aceptación de aquellas que trabajan con binarios dirigiendo esta investigación hacia el diseño de una herramienta que permite la automatización de la generación de los mismos para la distribución Nova.
- La investigación crítica sobre la gestión de paquetes binarios en la distribución Nova, permitió valorar su proceso de generación contribuyendo a la determinación de los requisitos de una herramienta que los automatice.
- La modelación realizada con la aplicación de la metodología XP y el Modelado Ágil (AM), validan los requerimientos de la herramienta determinados en la investigación.
- Se logra establecer el diseño de una herramienta que permitirá a la distribución Nova generar sus propios paquetes binarios a partir de los diversos repositorios de fuentes.

Recomendaciones

- Implementación de la herramienta diseñada en la investigación.
- Diseño de una interfaz web de administración de la aplicación, con los protocolos de seguridad requeridos para este tipo de sistemas.
- Diseño e implementación de métodos inteligentes para dar respuesta a los errores ocurridos durante el proceso de compilación de los paquetes de fuentes.

Bibliografía

Referencias Bibliográficas

1. Affolter, F., 2007: “Koji”, en línea: http://fedoraproject.org/wiki/Koji, Consultado: 24/03/2008.
2. Alegsa, 2008a: en línea: http://www.alegsa.com.ar/Dic/paquete%2520de%2520software.php+paquete+de+software+es&hl=es&ct=clnk&cd=3&gl=cu, Consultado: 24-03-2008
3. Alegsa, 2008b: en línea: http://www.alegsa.com.ar/Dic/sistema%2520de%2520gestion%2520de%2520paquetes.php+Gestor+de+paquetes&hl=es&ct=clnk&cd=14&gl=cu, consultado: 27-03-2008
4. Ambler, S, 2006a: “Flow Charts”, en línea: http://www.agilemodeling.com/artifacts/flowChart.htm, Consultado: 18/05/2008
5. Ambler, S., 2006b: “Class Responsability Colaborators (CRC) Models”, en línea: http://www.agilemodeling.com/artifacts/crcModel.htm, Consultado: 20/05/2008
6. Ambler, S. W., 2007: “An Introduction to Agile Modeling” en línea: http://www.agilemodeling.com/essays/introductionToAM.htm, Consultado: 31/03/2008
7. Ambler, S., 2008: “Agile Architecture: Strategies for Scaling Agile Development”, en línea: http://www.agilemodeling.com/essays/agileArchitecture.htm#Model, Consultado: 25/05/2008
8. Avizora, 2008: “Computación. Términos más comunes de la Industria de las Computadoras”, en línea: http://www.avizora.com/glosarios/glosarios_c/textos_c/computacion_industria_c_0007.htm, Consultado: 17/06/2008
9. Bailey, E. C., 2000: “Maximum RPM”, en línea: https://www.redhat.com/docs/books/max-rpm/max-rpm.pdf, Consultado: 28/03/2008
10. Barnes, D., 1999: “RPM Howto”, en línea: http://www.rpm.org/RPM-HOWTO/, Consultado: 28/03/2008
11. Barth, A., 2007: “Debian Developer's Reference”, en línea: http://www.debian.org/doc/manuals/developers-reference/ch-resources.en.html#s4.6.3, Consultado: 28/03/2008

12. Beck, K., 1999: “Extreme Programming Explained”
13. Burjans, L., 2007: “Nova, GNU/Linux cubano”, en línea: http://www.com-sl.org/article.php?story=linux-nova, Consultado: 20/03/2008
14. Cabanes, N., 2008: “Diccionario básico de Informática”, en línea: http://usuarios.lycos.es/Resve/diccioninform.htm#K, Consultado: 17/06/2008
15. Canós, J. H.; Letelier, P. y Penadés, M. C., 2003: “Metodologías Ágiles en el desarrollo de software”, en línea: , Consultado: 31/03/2008
16. CollabNet, 2008: “Welcome to ArgoUML”, en línea: http://argouml.tigris.org/, Consultado: 15/04/2008
17. Cplusplus, 2008: “A brief description”, en línea: http://www.cplusplus.com/info/description.html, Consultado: 16/04/2008
18. Definición, 2008a: “Definición de Repositorio”, en línea: http://www.definicion.org/repositorio, Consultado: 7/04/2008
19. Definición, 2008b: “Definición de xml”, en línea: http://www.definicion.org/xml, Consultado: 17/06/2008
20. Devja Vu, 2008: “Gaphor-UML modelling”, en línea: http://gaphor.devjavu.com/, Consultado: 15/04/2008
21.
22. Enciclopedia, 2008: “Lenguajes de programación”, en línea: http://enciclopedia.us.es/index.php/Lenguaje_de_programaci%C3%B3n#Lenguajes_de_alto_nivel, Consultado: 17/06/2008
23. Eric Python IDE, 2008: “What is Eric”, en línea: http://www.die-offenbachs.de/eric/index.html, Consultado: 31/03/2008
24. Fernández G., D., 2002: “Manejo básico de RPM”, en línea: http://www.elrincondelprogramador.com/default.asp?pag=articulos%2Fleer.asp&id=8, Consultado: 28/03/2008
25. Foster-Johnson, E., 2003: “Red Hat RPM Guide”, en línea: http://www.google.com.cu/books?id=L5hOQj-pQrwC&q=buildroot&dq=buildroot&pgis=1, Consultado: 17/06/2008

26. Fowler, M., 2004: “Is Design Dead?”, en línea: http://martinfowler.com/articles/designDead.html, Consultado: 25/05/2008
27. Free Software Foundation, 2002: “Bash Reference Manual”, en línea: http://www.gnu.org/software/bash/manual/bashref.html#Top, Consultado: 31/03/2008
28. Free Software Foundation, 2007a: “What is GNU?”, en línea: http://www.gnu.org/, Consultado: 20/03/2008
29. Free Software Foundation, 2007b: “GNU General Public License”, en línea: http://www.gnu.org/licenses/gpl.txt, Consultado: 17/06/2008
30. Free Software Foundation, 2007c: “Announcing ncurses 5.6”, en línea: http://www.gnu.org/software/ncurses/ncurses.html, Consultado: 17/06/2008
31. Gentoo Foundation, 2008a: “What is Gentoo?”, en línea: http://www.gentoo.org/main/en/about.xml, Consultado: 17/06/2008
32. Gentoo Foundation, 2008b: “A Portage Introduction”, en línea: http://www.gentoo.org/doc/en/handbook/handbook-x86.xml?part=2&chap=1, Consultado: 17/06/2008
33. Gentoo Linux, 2008a: “A Portage Introduction”, en línea: http://www.gentoo.org/doc/en/handbook/handbook-x86.xml?part=2&chap=1, Consultado: 28/03/2008
34. Gentoo Linux, 2008b: “Ebuild Howto”, en línea: http://www.gentoo.org/proj/en/devrel/handbook/handbook.xml?part=2&chap=1#doc_chap2, Consultado: 28/03/2008
35. GNOME Project, 2008: “Dia”, en línea: http://www.gnome.org/projects/dia/, Consultado: 15/04/2008
36. Goñi, A., 2007: “Nova LNX como plataforma de desarrollo personalizada.”
37. Gracia, J., 2006: “Gestión de proyectos con SCRUM”, en línea: http://www.ingenierosoftware.com/equipos/scrum.php, Consultado: 31/03/2008
38. Gray, B., 2008: “Glosario”, en línea: http://cutepaste.wordpress.com/glosario-y-definiciones/, Consultado: 17/06/2008
39. Gutierrez, I., 2001: “Glosario sobre Linux”, en línea: http://www.escomposlinux.org/glosario/#nucleo, Consultado: 17/06/2008

40. Gutiérrez, J. J.; Escalona, M. J.; Mejías, M. y J. Torres, 2008: “Pruebas del sistema en Programación Extrema.”
41. Herrera, A. y Rodríguez, Y., 2006: “Desarrollo y Mantenimiento de una distribución de Linux”
42. Jacobson, I.; Booch, G. y Rumbaugh, J., 2000: “El Proceso Unificado de Software.”
43. Java, 2008: “Learn About Java Technology”, en línea: http://www.java.com/en/about/, Consultado: 16/04/2008
44. Jeffries, R.; Anderson, A y Hendrickson, Ch., 2000: “Extreme Programming Installed”
45. Letelier, P. y Penadés, M. C., 2008: “Metodologías Ágiles en el desarrollo de software: eXtreme Programming (XP).”
46. Limodou, 2007: “Ulipad 3.6”, en línea: http://pypi.python.org/pypi/UliPad/3.6, Consultado: 31/03/2008
47. Linux Onlines Inc, 2007a: “What is Linux”, en línea: http://www.linux.org/info/index.html, Consultado: 20/03/2008
48. Linux Onlines Inc, 2007b: “Linux Distributions”, en línea: http://www.linux.org/dist/index.html, Consultado: 20/03/2008
49. Microsoft Corporation, 2002: “Glosario”, en línea: http://www.microsoft.com/technet/archive/visio/visio2002/plan/glossary.mspx?mfr=true, Consultado: 17/06/2008
50. Microsoft Corporation, 2003: “Microsoft Solution Framework version 3.0 Overview”, en línea: http://download.microsoft.com/download/3/a/6/3a6b7624-da2a-4883-a413-99281adadff3/MSF_v3_Overview%20Whitepaper.pdf, Consultado: 15/04/2008
51. Noronha, G., 2004: “APT HOWTO”, en línea: http://www.debian.org/doc/manuals/apt-howto/ch-sourcehandling.es.html, Consultado: 28/03/2008
52. OpenSuse, 2007: “Build Service”, en línea: http://es.opensuse.org/Build_Service, Consultado: 31/03/2008
53. Python, 2008: “Python Programming Language”, en línea: http://www.python.org/, Consultado: 31/03/2008
54. Red Hat Software Inc., 1999: “Utilizar RPM”, en línea: http://www.ibiblio.org/pub/linux/docs/LuCaS/Manuales-LuCAS/RHAT/rhl-ig-

6.0es/node269.html, Consultado: 28/03/2008

55. Reynoso, B., 2004: “Introducción a la Arquitectura de Software.”
56. Rizzo, P. M., 2007: “Ututo XS GNU 2006.0”, en línea: https://www.ututo.org/freewiki/index.php/UTUTO_XS_GNU_2006.0, Consultado: 31/03/2008
57. Sandoval, A., 2006: “¿Qué son los paquetes de software?”, en línea: http://www.microteknologias.cl/linux_paquetes.html, Consultado: 24/03/2008
58. Siever, E., Weber, A., 2005: “Linux in a Nutshell”, en línea: http://www.google.com.cu/books?id=M242s12pV5wC&pg=PA28&dq=NFS&sig=NJuYyU_Zi9xw0QqCK1yrb81pEzc, Consultado: 17/06/2008
59. Slackware Linux Inc., 2005: “Slackware Linux Essential”, en línea: http://www.slackbook.org/html/index.html, Consultado: 28/03/2008
60. Smith, C., 2006: “Linux NFS-Howto”, en línea: http://nfs.sourceforge.net/nfs-howto/ar01s02.html#whatis_nfs, Consultado: 17/06/2008
61. Stallman, R., 2007: “The GNU Project”, en línea: http://www.gnu.org/gnu/thegnuproject.html, Consultado: 20/03/2008
62. Tanenbaum, A, 1991: “Sistemas operativos: diseño e implementación”
63. Tejada Shoebridge S.L., 2002: “¿Qué es VPN? Glosario”, en línea: http://www.mundovpn.com/queesvpn_glo.html#c, Consultado: 17/06/2008
64. The Open Group, 2007: “Shell command language”, en línea: http://www.opengroup.org/onlinepubs/009695399/utilities/xcu_chap02.html, Consultado:17/06/2008
65. UserLand Software, 2008: “What is XML-RPC?”, en línea: http://www.xmlrpc.com/, Consultado:17/06/2008
66. Van Rossum, G. y Warsaw B., 2001: “Style Guide for Python Code”, en línea: http://www.python.org/dev/peps/pep-0008/, Consultado: 25/05/2008
67. Visual Paradigm, 2008: “Why Visual Paradigm for UML?”, en línea: http://www.visual-paradigm.com/product/vpuml/, Consultado: 31/03/2008
68. Webopedia, 2008: “software package”, en línea: http://www.webopedia.com/TERM/S/software_package.html, Consultado: 24/03/2008
69. Wikipedia, 2007: en línea: , Consultado:

70. Wikipedia, 2008a: “Distribución Linux”, en línea: http://es.wikipedia.org/wiki/Distribuci%C3%B3n_Linux, Consultado: 20/03/2008
71. Wikipedia, 2008b: “Paquete de software”, en línea: http://es.wikipedia.org/wiki/Paquetes_de_software, Consultado: 24/03/2008
72. Wikipedia, 2008c: “Dpkg”, en línea: http://es.wikipedia.org/wiki/Dpkg, Consultado: 28/03/2008
73. Wikipedia, 2008d: “Advanced Packaging Tool”, en línea: http://es.wikipedia.org/wiki/Advanced_Packaging_Tool, Consultado: 28/03/2008
74. Wikipedia, 2008e: “Slapt get”, en línea: http://es.wikipedia.org/wiki/Slapt-get, Consultado: 28/03/2008
75. Wikipedia, 2008f: “Scrum”, en línea: http://es.wikipedia.org/wiki/Scrum, Consultado: 31/03/2008
76. Wikipedia, 2008g: “C++”, en línea: http://es.wikipedia.org/wiki/C%2B%2B, Consultado: 31/03/2008
77. Wikipedia, 2008h: “Lenguaje de programación Java”, en línea: http://es.wikipedia.org/wiki/Lenguaje_de_programaci%C3%B3n_Java, Consultado: 31/03/2008
78. Wikipedia, 2008i: “Python”, en línea: http://es.wikipedia.org/wiki/Python, Consultado: 31/03/2008
79. Wikipedia, 2008j: “Boa Constructor”, en línea: http://en.wikipedia.org/wiki/Boa_Constructor, Consultado: 31/03/2008
80. Wikipedia, 2008k: “Integrated development environment”, en línea: http://en.wikipedia.org/wiki/Integrated_Development_Environment, Consultado: 17/06/2008
81. Wikipedia, 2008l: “Application programming interface”, en línea: http://en.wikipedia.org/wiki/API, Consultado: 17/06/2008
82. Wikipedia, 2008m: “Configuration file”, en línea: http://en.wikipedia.org/wiki/Configuration_file, Consultado: 17/06/2008
83. Wikipedia, 2008n: “Firma digital”, en línea: http://es.wikipedia.org/wiki/Firma_digital, Consultado: 17/06/2008
84. Wikipedia, 2008o: “GTK+”, en línea: http://es.wikipedia.org/wiki/GTK+, Consultado: 17/06/2008
85. Wikipedia, 2008p: “Kernel (computer science)”, en línea: http://en.wikipedia.org/wiki/Kernel_%28computer_science%29, Consultado: 17/06/2008

86. Wikipedia, 2008q: “MD5”, en línea: http://en.wikipedia.org/wiki/MD5, Consultado: 17/06/2008
87. wxPython, 2008: “wxPython”, en línea: http://www.wxpython.org/, Consultado: 17/06/2008
88. ZonaSiete, 2004: “Manual Linux eminentemente práctico, ZonaSiete.ORG”, en línea: http://www.zonasiete.org/manual/ch01s03.html, Consultado: 24/03/2008
89. Zuser, W.; Heil, S. y Grechenig, T., 2008: “Software quality development and assurance in RUP, MSF and XP: a comparative study”, en línea: http://portal.acm.org/citation.cfm?id=1083300&dl=acm&coll=, Consultado: 16/04/2008

Bibliografía Consultada

1. Alegsa, 2008: ,en línea: ,Consultado: 27/03/2008
2. Ambler, S., 2008: “Agile Modeling (AM) Home Page”, en línea: http://www.agilemodeling.com/
3. Brouchier, Julien, 1999: “Manual de Red Hat”, en línea: http://www.ibiblio.org/pub/linux/docs/LuCaS/Manuales-LuCAS/RHAT/rhl-ig-6.0es/node267.htm, Consultado: 27/03/2008
4. CosasLibres, 2004: “Software”, en línea: http://www.cosaslibres.com/software.html, Consultado: 25/03/2008
5. DebianTalkImporter, 2005: “Una balanza entre Debian y Gentoo”, en línea: http://tralla.org/debian/archives/072358.html, Consultado: 27/03/2008
6. Dsic, 2008: “Ejemplo de desarrollo software utilizando la metodología XP”, en línea: http://www.dsic.upv.es/asignaturas/facultad/lsi/ejemploxp/Gestion_Proyecto.html#plan_inicial, consulta última: 30/05/2008
7. El Santo, 2006: “Primeros pasos en Debian: apt-get, dpkg, Aptitude y Synaptic”, en línea: http://www.ecualug.org/?q=2006/01/30/blog/elsanto/primeros_pasos_en_debian_apt_get_dpkg_aptitude_y_synaptic, Consultado: 27/03/2008
8. Fernando-Sanguino P., J., 2001: “Herramientas de gestión de paquetes para Debian”, en línea: http://www.debian.org/international/spanish/contrib/paqifaz.html, Consultado: 27/03/2008
9. Fowler, M., 2003: “La Nueva Metodología”, en línea: http://www.programacionextrema.org/articulos/newMethodology.es.html#tth_sEc5, Consultado: 29/03/2008
10. Geany, 2008: “About Geany”, en línea: http://geany.uvena.de/Main/About, Consultado: 31/03/2008
11. Gentoo, 2008a: “Gentoo Development Guide”, en línea: http://devmanual.gentoo.org/general-concepts/tree/index.html, Consultado: 27/03/2008
12. Gentoo, 2008b: “Portage Features”, en línea: http://www.gentoo.org/doc/en/handbook/2008.0/handbook-x86.xml?part=2&chap=3, consulta última: 30/05/2008

13. GNU, 2007: “GNU Emacs”, en línea: http://www.gnu.org/software/emacs/, Consultado: 31/03/2008
14. GujUlat, 2007: ,en línea: ,Consultado: 27/03/2008
15. NetBeans, 2008: “Setting Up Projects”, en línea: http://www.netbeans.org/kb/55/using-netbeans/project_setup.html#pgfId-1020595, Consultado: 29/03/2008
16. Python, 2008a: “Eric”, en línea: http://wiki.python.org/moin/eric, Consultado: 31/03/2008
17. Python, 2008b: “Boa Constructor”, en línea: http://wiki.python.org/moin/BoaConstructor, Consultado: 31/03/2008
18. Ramos, R., 2002: “apt y dpkg. Herramientas de gestión de paquetes para Debian”, en línea: http://www.linuca.org/body.phtml?nIdNoticia=39, Consultado: 27/03/2008
19. Riveros, O., 2007: “¿Qué es un gestor de paquetes?”, en línea: http://www.entrebits.cl/foros/linux/8274-que-es-un-gestor-de-paquetes.html, Consultado: 27/03/2008
20. Shoemaker, K., 2008: “Flipping the Linux switch: Package management 101”, en línea: http://www.downloadsquad.com/2008/01/08/flipping-the-linux-switch-package-management-101/, Consultado: 27/03/2008
21. SourceFourge, 2008: “Boa Constructor”, en línea: http://boa-constructor.sourceforge.net/, Consultado: 31/03/2008
22. TrabeSoluciones, 2008: “Desarrollo Ágil”, en línea: http://www.trabesoluciones.com/es/development/methodology, Consultado: 29/03/2008
23. Wikipedia, 2008a: Linux (núcleo), en línea: http://es.wikipedia.org/wiki/Linux_%28n%C3%BAcleo%29, Consultado: 18/03/2008
24. Wikipedia, 2008b: GNU, en línea: http://es.wikipedia.org/wiki/Proyecto_GNU, Consultado: 18/03/2008
25. Wikipedia, 2008c: GNU/Linux, en línea: http://es.wikipedia.org/wiki/GNU/Linux, Consultado: 18/03/2008
26. Wikipedia, 2008d: Linux, en línea: http://es.wikipedia.org/wiki/Linux, Consultado: 18/03/2008
27. Wikipedia, 2008e: Anexo: Distribuciones Linux, en línea: http://es.wikipedia.org/wiki/Anexo:Lista_de_Distribuciones_Linux, Consultado: 18/03/2008

28. Wikipedia, 2008f: “Sistema de gestión de paquetes”, en línea: http://es.wikipedia.org/wiki/Gestor_de_paquetes, Consultado: 27/03/2008
29. Wikipedia, 2008g: “Sistema de gestión de paquetes”, en línea: http://es.wikipedia.org/wiki/Sistema_de_gesti%C3%B3n_de_paquetes, Consultado: 27/03/2008
30. Wikipedia, 2008h: “NetBeans”, en línea: http://es.wikipedia.org/wiki/NetBeans, Consultado: 29/03/2008
31. Wikipedia, 2008i: “Eric Python IDE”, en línea: http://en.wikipedia.org/wiki/Eric_Python_IDE, Consultado: 31/03/2008
32. wxPython, 2008: “Boa Constructor”, en línea: http://wiki.wxpython.org/Boa%20Constructor, Consultado: 31/03/2008

Anexos

Anexo 1: Historias de Usuarios

Historias de Usuarios		
Nombre de la Historia: Generar paquetes de código binario.	Fecha: 6/Marzo/2008	Versión: 1.0
No. de la Historia: 1	Dependencia: -	Prioridad(Alta/Media/Baja): Alta
Descripción: Según la lista indicada compilar los paquetes de código fuente con sus dependencias para las arquitecturas deseadas dentro de las que se mostrarán en pantalla. En caso de que no se seleccione ninguna arquitectura de las presentadas se generarán los paquetes binarios para cada una de ellas. Observaciones: La lista indicada: es un fichero de texto que puede estar en el sistema conteniendo los paquetes que faltan por generar. También se refiere a una lista que puede crear el usuario de los paquetes que necesite generar.		

Historias de Usuarios		
Nombre de la Historia: Gestionar errores.	Fecha: 6/Marzo/2008	Versión: 1.0
No. de la Historia: 2	Dependencia: HU 1	Prioridad(Alta/Media/Baja): Alta
Descripción: Durante el proceso de generación de los paquetes binarios el sistema debe monitorear los fallos ocurridos. Deben validarse los errores más comunes para que el sistema pueda darle solución. En caso de que el sistema no pueda resolver el fallo debe generar un reporte sobre lo ocurrido para que pueda ser Consultado y analizado posteriormente. Luego debe continuar el proceso a partir de la tarea de compilación que sigue a la fallida, donde se interrumpió, para que el resto de los paquetes sean generados. Observaciones: Fallos o Errores: problemas al compilar algún paquete, que puede ser por estar		

corrupto el paquete o por faltar alguna de las dependencias. Problemas de interrupción de la compilación.

Historias de Usuarios			
Nombre de la Historia: Generar informes.		Fecha: 6/Marzo/2008	Versión: 1.0
No. de la Historia: 3	Dependencia: HU 2	Prioridad(Alta/Media/Baja): Media	
Descripción: Siempre que el sistema termine de generar un grupo de paquetes binarios debe generar un informe de estado describiendo el cumplimiento de las tareas. En caso de la existencia de un error se generará un informe que contenga el estado en que se encontraba el proceso. Además de generar un informe con los fallos ocurridos durante la compilación de un grupo de paquetes. Observaciones: Estado del proceso de generación de paquetes binarios: último paquete compilado correctamente, tiempo, arquitecturas para las que se compiló, entre otros.			

Anexo 2: Plan de Release

Release	Descripción de la iteración	Orden de las HU a implementar	Duración total (días)
1	Entregar la herramienta con las funcionalidades esenciales, lo que incluye las dos primeras historias de usuario. Será evaluada por el usuario al finalizar, para la obtención de una herramienta con la calidad requerida por el cliente.	● HU 1: Generar paquetes de código binario. ● HU 2: Gestionar errores.	21
2	Se pretende refinar el sistema con funcionalidades adicionales como la expuesta en la tercera historia de usuario. Cuando se termine la implementación, será evaluada por el cliente para ser entregada con la calidad requerida.	● HU 3: Generar reportes.	10

Anexo 3: Tareas de Ingeniería

Estas estan ordenadas por historia de usuario con la que se relacionan y por iteraciones:

Iteración 1:

Tareas de Ingeniería	
Número de la Tarea: 1	Número de la Historia de Usuario: 1
Nombre de la Tarea: Diseño de la interfaz “Generar paquetes”	
Fecha de inicio:	Fecha de fin:
Desarrollador Responsable: Mónica Ma. Albo Castro	
Descripción: Debe elaborarse una interfaz cómoda para que el usuario pueda seleccionar la opción a la que desee acceder. Además deben diseñarse las interfaces que mostrarán las distintas opciones de la configuración, en caso de que el usuario determine usar esa opción.	

Tareas de Ingeniería	
Número de la Tarea: 2	Número de la Historia de Usuario: 1
Nombre de la Tarea: Introducción de la configuración para la compilación.	
Fecha de inicio:	Fecha de fin:
Desarrollador Responsable: Mónica Ma. Albo Castro	
Descripción: Debe mostrarse las opciones de configuración para la compilación, como son las arquitecturas de hardware para las cuales se generarán los paquetes, la lista de estos a usar durante el proceso. Además debe cargarse el fichero de configuración de manera que pueda ser utilizado cómodamente durante la compilación.	

Tareas de Ingeniería	
Número de la Tarea: 3	Número de la Historia de Usuario: 1
Nombre de la Tarea: Compilación de la lista de paquetes.	
Fecha de inicio:	Fecha de fin:
Desarrollador Responsable: Mónica Ma. Albo Castro	
Descripción: Dada la lista de los paquetes de fuentes en la configuración se obtendrá la lista de las dependencias, estas deben ordenarse de manera que puedan compilarse independiente, sin dependencias.	

Tareas de Ingeniería	
Número de la Tarea: 1	Número de la Historia de Usuario: 2
Nombre de la Tarea: Descargar paquetes de fuentes del repositorio.	
Fecha de inicio:	Fecha de fin:
Desarrollador Responsable: Mónica Ma. Albo Castro	
Descripción: El sistema debe conectarse al repositorio y buscar los paquete de fuente que se desea compilar. En caso de que los encuentre debe verificarse si hubo alguna falla por espacio insuficiente en disco o por estar el archivo corrupto antes de descargar los paquetes.	

Tareas de Ingeniería	
Número de la Tarea: 2	Número de la Historia de Usuario: 2
Nombre de la Tarea: Comprobar proceso de compilación	
Fecha de inicio:	Fecha de fin:
Desarrollador Responsable: Mónica Ma. Albo Castro	
Descripción: Durante la compilación debe chequearse los paquetes marcados como Mask y registrarse conjunto con cualquier otro error ocurrido durante el proceso.	

Tareas de Ingeniería	
Número de la Tarea: 3	Número de la Historia de Usuario: 2
Nombre de la Tarea: Subir paquete binario al servidor.	
Fecha de inicio:	Fecha de fin:
Desarrollador Responsable: Mónica Ma. Albo Castro	
Descripción: Debe establecerse una conexión con el servidor para llevar a cabo la autenticación, si no falla se copiarán el paquete en la categoría correspondiente.	

Iteración 2:

Tareas de Ingeniería	
Número de la Tarea: 1	Número de la Historia de Usuario: 3
Nombre de la Tarea: Generar informe de estado.	
Fecha de inicio:	Fecha de fin:
Desarrollador Responsable: Mónica Ma. Albo Castro	
Descripción: Para generar el informe de estado debe chequearse la fecha del último generado, para comenzar a buscar los logs a partir de esa fecha. En caso de que se encuentren se creará un documento donde copiar los datos brindados por los logs, quedando así conformado el informe.	

Tareas de Ingeniería	
Número de la Tarea: 2	Número de la Historia de Usuario: 3
Nombre de la Tarea: Generar informe de errores.	
Fecha de inicio:	Fecha de fin:
Desarrollador Responsable: Mónica Ma. Albo Castro	
Descripción: Primero se chequeará la fecha del último informe generado para buscar los logs de errores a partir de esa fecha, si existen estos logs se copiará su contenido en un documento nuevo.	

Anexo 4: Casos de Pruebas de Aceptación

Estos también fueron ordenados por iteración:

Iteración 1:

Pruebas de Aceptación	
Número del Caso de Prueba: 1	Número de la Historia de Usuario: 1
Nombre del Caso de Prueba: Definición de la arquitectura de hardware.	
Entradas: En este caso no se seleccionará ninguna de las arquitecturas de hardware brindadas por el sistema.	
Resultado Esperado: El sistema debe continuar su ejecución generando los paquetes para todas las arquitecturas como una opción predeterminada.	
Observaciones:	

Pruebas de Aceptación	
Número del Caso de Prueba: 2	Número de la Historia de Usuario: 1
Nombre del Caso de Prueba: Selección de arquitecturas de hardware.	
Entradas: Se marcará varias arquitecturas de hardware de las brindadas por el sistema seleccionadas aleatoriamente.	
Resultado Esperado: Se compilarán los paquetes de fuentes para las arquitecturas seleccionadas.	
Observaciones: Este caso de prueba debe realizarse varias veces, con varias combinaciones.	

Pruebas de Aceptación	
Número del Caso de Prueba: 3	Número de la Historia de Usuario: 1
Nombre del Caso de Prueba: Definir lista de paquetes a compilar.	
Entradas: Determinar el uso de la lista que tiene el sistema predeterminada con los paquetes que	

faltan por compilar. Resultado Esperado: Deben ser compilados los paquetes definidos en la lista predeterminada. Observaciones:

Pruebas de Aceptación	
Número del Caso de Prueba: 4	Número de la Historia de Usuario: 1
Nombre del Caso de Prueba: Seleccionar una lista nueva. Entradas: Seleccionar la opción que permite al usuario definir la lista de paquetes a generar. Resultado Esperado: Mostrar un editor de texto, de fácil manejo, que permita al usuario introducir el nombre de los paquetes que desea generar. Observaciones:	

Pruebas de Aceptación	
Número del Caso de Prueba: 5	Número de la Historia de Usuario: 1
Nombre del Caso de Prueba: Introducir nombres no válidos de paquetes	
Entradas: Introducir en la lista nombres no válidos de los paquetes que se desean compilar.	
Resultado Esperado: El sistema al guardar el documento creado con la lista debe chequear que todos los nombres sean válidos, en caso de que no sea así mostrar un mensaje indicando el error.	
Observaciones:	

Pruebas de Aceptación	
Número del Caso de Prueba: 6	Número de la Historia de Usuario: 1
Nombre del Caso de Prueba: Introducir correctamente los nombres de los paquetes.	
Entradas: Introducir en la lista los nombres de los paquetes que se desean compilar correctamente.	
Resultado Esperado: Guardar el documento y pasar a chequear el repositorio donde el sistema debe buscar esos paquetes.	
Observaciones:	

Pruebas de Aceptación	
Número del Caso de Prueba: 7	Número de la Historia de Usuario: 1
Nombre del Caso de Prueba: Dirección de repositorio incorrecta.	
Entradas: Introducir una dirección de repositorio con datos no válidos.	
Resultado Esperado: El sistema debe mostrar un mensaje indicando que el repositorio indicado no es correcto.	
Observaciones:	

Pruebas de Aceptación	
Número del Caso de Prueba: 8	Número de la Historia de Usuario: 1
Nombre del Caso de Prueba: Introducir dirección de repositorio.	
Entradas: Introducir una dirección valida para el repositorio en el cual el sistema buscará los paquetes a compilar.	
Resultado Esperado: El sistema proseguirá a compilar los paquetes encontrados en el repositorio indicado.	
Observaciones:	

Pruebas de Aceptación	
Número del Caso de Prueba: 9	Número de la Historia de Usuario: 1
Nombre del Caso de Prueba: Rectificar repositorio.	
Entradas: No introducir ninguna dirección de repositorio.	
Resultado Esperado: El sistema utilizará el repositorio que tiene configurado de manera predeterminada, mostrando un mensaje al usuario con esta información.	
Observaciones:	

Iteración 2:

Pruebas de Aceptación	
Número del Caso de Prueba: 1	Número de la Historia de Usuario: 3
Nombre del Caso de Prueba: Solicitar informe de estado.	
Entradas: Marcar la opción de generar informe de estado. Resultado Esperado: El sistema debe mostrar un informe con todo lo ocurrido durante el proceso de compilación.	
Observaciones: Este caso de prueba debe realizarse 2 veces para verificar que si no ha habido compilaciones desde el último informe no se genere uno nuevo, se muestre un mensaje.	

Pruebas de Aceptación	
Número del Caso de Prueba: 2	Número de la Historia de Usuario: 3
Nombre del Caso de Prueba: Solicitar informe de errores.	
Entradas: Seleccionar la opción de generar informe de los errores ocurridos.	
Resultado Esperado: Debe mostrarse un informe con los errores ocurridos durante la compilación, con el dato de si fueron solucionados o no.	
Observaciones: Debe evaluarse este caso dos veces para verificar que si no ha habido compilaciones desde el último informe no se genere uno nuevo, se muestre un mensaje.	

Anexo 5: Estándar de Programación para Python PEP 80

Introduction

This document gives coding conventions for the Python code comprising the standard library in the main Python distribution. Please see the companion informational PEP describing style guidelines for the C code in the C implementation of Python[1].

This document was adapted from Guido's original Python Style Guide essay[2], with some additions from Barry's style guide[5]. Where there's conflict, Guido's style rules for the purposes of this PEP. This PEP may still be incomplete (in fact, it may never be finished <wink>).

A Foolish Consistency is the Hobgoblin of Little Minds

One of Guido's key insights is that code is read much more often than it is written. The guidelines provided here are intended to improve the readability of code and make it consistent across the wide spectrum of Python code. As PEP 20 [6] says, "Readability counts".

A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is most important.

But most importantly: know when to be inconsistent -- sometimes the style guide just doesn't apply. When in doubt, use your best judgment. Look at other examples and decide what looks best. And don't hesitate to ask!

Two good reasons to break a particular rule:

(1) When applying the rule would make the code less readable, even for someone who is used to reading code that follows the rules.

(2) To be consistent with surrounding code that also breaks it (maybe for historic reasons) -- although this is also an opportunity to clean up someone else's mess (in true XP style).

Code lay-out

Indentation

Use 4 spaces per indentation level.

For really old code that you don't want to mess up, you can continue to use 8-space tabs.

Tabs or Spaces?

Never mix tabs and spaces.

The most popular way of indenting Python is with spaces only. The second-most popular way is with tabs only. Code indented with a mixture of tabs and spaces should be converted to using spaces exclusively. When invoking the Python command line interpreter with the -t option, it issues warnings about code that illegally mixes tabs and spaces. When using -tt these warnings become errors. These options are highly recommended!

For new projects, spaces-only are strongly recommended over tabs. Most editors have features that make this easy to do.

Maximum Line Length

Limit all lines to a maximum of 79 characters.

There are still many devices around that are limited to 80 character lines; plus, limiting windows to 80 characters makes it possible to have several windows side-by-side. The default wrapping on such devices disrupts the visual structure of the code, making it more difficult to understand. Therefore, please limit all lines to a maximum of 79 characters. For flowing long blocks of text (docstrings or comments), limiting the length to 72 characters is recommended.

The preferred way of wrapping long lines is by using Python's implied line continuation inside parentheses, brackets and braces. If necessary, you can add an extra pair of parentheses around an expression, but

sometimes using a backslash looks better. Make sure to indent the continued line appropriately. Some examples:

```
class Rectangle(Blob):

    def __init__(self, width, height,
                 color='black', emphasis=None, highlight=0):
        if width == 0 and height == 0 and \
           color == 'red' and emphasis == 'strong' or \
           highlight > 100:
            raise ValueError("sorry, you lose")
        if width == 0 and height == 0 and (color == 'red' or
                                           emphasis is None):
            raise ValueError("I don't think so")
        Blob.__init__(self, width, height,
                      color, emphasis, highlight)
```

Blank Lines

Separate top-level function and class definitions with two blank lines.

Method definitions inside a class are separated by a single blank line.

Extra blank lines may be used (sparingly) to separate groups of related functions. Blank lines may be

omitted between a bunch of related one-liners (e.g. a set of dummy implementations).

Use blank lines in functions, sparingly, to indicate logical sections.

Python accepts the control-L (i.e. ^L) form feed character as whitespace; Many tools treat these characters as page separators, so you may use them to separate pages of related sections of your file.

Encodings (PEP 263**)**

Code in the core Python distribution should aways use the ASCII or Latin-1 encoding (a.k.a. ISO-8859-1). For Python 3.0 and beyond, UTF-8 is preferred over Latin-1, see PEP 3120.

Files using ASCII (or UTF-8, for Python 3.0) should not have a coding cookie. Latin-1 (or UTF-8) should only be used when a comment or docstring needs to mention an author name that requires Latin-1; otherwise, using \x, \u or \U escapes is the preferred way to include non-ASCII data in string literals.

For Python 3.0 and beyond, the following policy is prescribed for the standard library (see PEP 3131): All identifiers in the Python standard library MUST use ASCII-only identifiers, and SHOULD use English words wherever feasible (in many cases, abbreviations and technical terms are used which aren't English). In addition, string literals and comments must also be in ASCII. The only exceptions are (a) test cases testing the non-ASCII features, and (b) names of authors. Authors whose names are not based on the latin alphabet MUST provide a latin transliteration of their names.

Open source projects with a global audience are encouraged to adopt a similar policy.

Imports

- Imports should usually be on separate lines, e.g.:

Yes: import os

import sys

No: import sys, os

    it's okay to say this though:

from subprocess import Popen, PIPE

- Imports are always put at the top of the file, just after any module comments and docstrings, and before module globals and constants.

Imports should be grouped in the following order:

1. standard library imports

2. related third party imports

3. local application/library specific imports

You should put a blank line between each group of imports.

Put any relevant __all__ specification after the imports.

- Relative imports for intra-package imports are highly discouraged. Always use the absolute package path for all imports. Even now that PEP 328 [7] is fully implemented in Python 2.5, its style of explicit relative imports is actively discouraged; absolute imports are more portable and usually more readable.

- When importing a class from a class-containing module, it's usually okay to spell this

from myclass import MyClass

from foo.bar.yourclass import YourClass

If this spelling causes local name clashes, then spell them

import myclass

import foo.bar.yourclass

and use "myclass.MyClass" and "foo.bar.yourclass.YourClass"

Whitespace in Expressions and Statements

Avoid extraneous whitespace in the following situations:

- Immediately inside parentheses, brackets or braces.

Yes: spam(ham[1], {eggs: 2})

No: spam(ham[1], { eggs: 2 })

- Immediately before a comma, semicolon, or colon:

Yes: if x == 4: print x, y; x, y = y, x

No: if x == 4 : print x , y ; x , y = y , x

- Immediately before the open parenthesis that starts the argument list of a function call:

Yes: spam(1)

No: spam (1)

- Immediately before the open parenthesis that starts an indexing or slicing:

Yes: dict['key'] = list[index]

No: dict ['key'] = list [index]

- More than one space around an assignment (or other) operator to align it with another.

Yes:

```
x = 1

y = 2

long_variable = 3
```

No:

```
x           = 1

y           = 2

long_variable = 3
```

Other Recommendations

- Always surround these binary operators with a single space on either side: assignment (=), augmented assignment (+=, -= etc.), comparisons (==, <, >, !=, <>, <=, >=, in, not in, is, is not), Booleans (and, or, not).

- Use spaces around arithmetic operators:

Yes:

```
i = i + 1

submitted += 1

x = x * 2 - 1

hypot2 = x * x + y * y

c = (a + b) * (a - b)
```

No:

```
i=i+1

submitted +=1

x = x*2 - 1

hypot2 = x*x + y*y

c = (a+b) * (a-b)
```

- Don't use spaces around the '=' sign when used to indicate a keyword argument or a default parameter value.

Yes:

```
def complex(real, imag=0.0):

    return magic(r=real, i=imag)
```

No:

```
def complex(real, imag = 0.0):

    return magic(r = real, i = imag)
```

- Compound statements (multiple statements on the same line) are generally discouraged.

Yes:

```
if foo == 'blah':

    do_blah_thing()

do_one()
```

```
do_two()

do_three()
```

Rather not:

```
if foo == 'blah': do_blah_thing()

    do_one(); do_two(); do_three()
```

- While sometimes it's okay to put an if/for/while with a small body on the same line, never do this for multi-clause statements. Also avoid folding such long lines!

Rather not:

```
if foo == 'blah': do_blah_thing()

for x in lst: total += x

while t < 10: t = delay()
```

Definitely not:

```
if foo == 'blah': do_blah_thing()

else: do_non_blah_thing()

try: something()

finally: cleanup()

do_one(); do_two(); do_three(long, argument,

                             list, like, this)
```

```
if foo == 'blah': one(); two(); three()
```

Comments

Comments that contradict the code are worse than no comments. Always make a priority of keeping the comments up-to-date when the code changes!

Comments should be complete sentences. If a comment is a phrase or sentence, its first word should be capitalized, unless it is an identifier that begins with a lower case letter (never alter the case of identifiers!).

If a comment is short, the period at the end can be omitted. Block comments generally consist of one or more paragraphs built out of complete sentences, and each sentence should end in a period.

You should use two spaces after a sentence-ending period.

When writing English, Strunk and White apply.

Python coders from non-English speaking countries: please write your comments in English, unless you are 120% sure that the code will never be read by people who don't speak your language.

Block Comments

Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code. Each line of a block comment starts with a # and a single space (unless it is indented text inside the comment).

Paragraphs inside a block comment are separated by a line containing a single #.

Inline Comments

Use inline comments sparingly.

An inline comment is a comment on the same line as a statement. Inline comments should be separated by at least two spaces from the statement. They should start with a # and a single space.

Inline comments are unnecessary and in fact distracting if they state the obvious. Don't do this:

```
x = x + 1                 # Increment x
```

But sometimes, this is useful:

```
x = x + 1                 # Compensate for border
```

Documentation Strings

Conventions for writing good documentation strings (a.k.a. "docstrings") are immortalized in PEP 257 [3].

- Write docstrings for all public modules, functions, classes, and methods. Docstrings are not necessary for non-public methods, but you should have a comment that describes what the method does. This comment should appear after the "def" line.

- PEP 257 describes good docstring conventions. Note that most importantly, the """ that ends a multiline docstring should be on a line by itself, and preferably preceded by a blank line, e.g.:

```
"""Return a foobang

    Optional plotz says to frobnicate the bizbaz first.

"""
```

- For one liner docstrings, it's okay to keep the closing """ on the same line.

Version Bookkeeping

If you have to have Subversion, CVS, or RCS crud in your source file, do it as follows.

```
__version__ = "$Revision: 60919 $"

# $Source$
```

These lines should be included after the module's docstring, before any other code, separated by a blank line above and below.

Naming Conventions

The naming conventions of Python's library are a bit of a mess, so we'll never get this completely consistent -- nevertheless, here are the currently recommended naming standards. New modules and packages (including third party frameworks) should be written to these standards, but where an existing library has a different style, internal consistency is preferred.

Descriptive: Naming Styles

There are a lot of different naming styles. It helps to be able to recognize what naming style is being used, independently from what they are used for.

The following naming styles are commonly distinguished:

- b (single lowercase letter)

- B (single uppercase letter)

- lowercase

- lower_case_with_underscores

- UPPERCASE

- UPPER_CASE_WITH_UNDERSCORES

- CapitalizedWords (or CapWords, or CamelCase -- so named because of the bumpy look of its letters[4]). This is also sometimes known as StudlyCaps.

Note: When using abbreviations in CapWords, capitalize all the letters of the abbreviation. Thus HTTPServerError is better than HttpServerError.

- mixedCase (differs from CapitalizedWords by initial lowercase character!)

- Capitalized_Words_With_Underscores (ugly!)

There's also the style of using a short unique prefix to group related names together. This is not used much in Python, but it is mentioned for completeness. For example, the os.stat() function returns a tuple whose items traditionally have names like st_mode, st_size, st_mtime and so on. (This is done to emphasize the correspondence with the fields of the POSIX system call struct, which helps programmers familiar with that.)

The X11 library uses a leading X for all its public functions. In Python, this style is generally deemed unnecessary because attribute and method names are prefixed with an object, and function names are prefixed with a module name.

In addition, the following special forms using leading or trailing underscores are recognized (these can generally be combined with any case convention):

- _single_leading_underscore: weak "internal use" indicator. E.g. "from M import *" does not import objects whose name starts with an underscore.

- single_trailing_underscore_: used by convention to avoid conflicts with Python keyword, e.g.

    Tkinter.Toplevel(master, class_='ClassName')

- __double_leading_underscore: when naming a class attribute, invokes name mangling (inside class FooBar, __boo becomes _FooBar__boo; see below).

- __double_leading_and_trailing_underscore__: "magic" objects or attributes that live in user-controlled namespaces. E.g. __init__, __import__ or __file__. Never invent such names; only use them as documented.

Prescriptive: Naming Conventions

Names to Avoid

Never use the characters `l' (lowercase letter el), `O' (uppercase letter oh), or `I' (uppercase letter eye) as single character variable names.

In some fonts, these characters are indistinguishable from the numerals one and zero. When tempted to use `l', use `L' instead.

Package and Module Names

Modules should have short, all-lowercase names. Underscores can be used in the module name if it improves readability. Python packages should also have short, all-lowercase names, although the use of underscores is discouraged.

Since module names are mapped to file names, and some file systems are case insensitive and truncate long names, it is important that module names be chosen to be fairly short -- this won't be a problem on Unix, but it may be a problem when the code is transported to older Mac or Windows versions, or DOS.

When an extension module written in C or C++ has an accompanying Python module that provides a higher level (e.g. more object oriented) interface, the C/C++ module has a leading underscore (e.g. _socket).

Class Names

Almost without exception, class names use the CapWords convention. Classes for internal use have a leading underscore in addition.

Exception Names

Because exceptions should be classes, the class naming convention applies here. However, you should use the suffix "Error" on your exception names (if the exception actually is an error).

Global Variable Names

(Let's hope that these variables are meant for use inside one module only.) The conventions are about the same as those for functions.

Modules that are designed for use via "from M import *" should use the __all__ mechanism to prevent exporting globals, or use the older convention of prefixing such globals with an underscore (which you might want to do to indicate these globals are "module non-public").

Function Names

Function names should be lowercase, with words separated by underscores as necessary to improve readability.

mixedCase is allowed only in contexts where that's already the prevailing style (e.g. threading.py), to retain backwards compatibility.

Function and method arguments

Always use 'self' for the first argument to instance methods.

Always use 'cls' for the first argument to class methods.

If a function argument's name clashes with a reserved keyword, it is generally better to append a single trailing underscore rather than use an abbreviation or spelling corruption. Thus "print_" is better than "prnt". (Perhaps better is to avoid such clashes by using a synonym.)

Method Names and Instance Variables

Use the function naming rules: lowercase with words separated by underscores as necessary to improve readability.

Use one leading underscore only for non-public methods and instance variables.

To avoid name clashes with subclasses, use two leading underscores to invoke Python's name mangling rules.

Python mangles these names with the class name: if class Foo has an attribute named __a, it cannot be accessed by Foo.__a. (An insistent user could still gain access by calling Foo._Foo__a.) Generally, double leading underscores should be used only to avoid name conflicts with attributes in classes designed to be subclassed.

Note: there is some controversy about the use of __names (see below).

Designing for inheritance

Always decide whether a class's methods and instance variables (collectively: "attributes") should be public or non-public. If in doubt, choose non-public; it's easier to make it public later than to make a public attribute non-public.

Public attributes are those that you expect unrelated clients of your class to use, with your commitment to avoid backward incompatible changes. Non-public attributes are those that are not intended to be used by third parties; you make no guarantees that non-public attributes won't change or even be removed.

We don't use the term "private" here, since no attribute is really private in Python (without a generally unnecessary amount of work).

Another category of attributes are those that are part of the "subclass API" (often called "protected" in other languages). Some classes are designed to be inherited from, either to extend or modify aspects of the class's behavior. When designing such a class, take care to make explicit decisions about which attributes are public, which are part of the subclass API, and which are truly only to be used by your base class.

With this in mind, here are the Pythonic guidelines:

- Public attributes should have no leading underscores.

- If your public attribute name collides with a reserved keyword, append a single trailing underscore to your attribute name. This is preferable to an abbreviation or corrupted spelling. (However, notwithstanding this rule, 'cls' is the preferred spelling for any variable or argument which is known to be a class, especially the first argument to a class method.)

Note 1: See the argument name recommendation above for class methods.

- For simple public data attributes, it is best to expose just the attribute name, without complicated accessor/mutator methods. Keep in mind that Python provides an easy path to future enhancement, should you find that a simple data attribute needs to grow functional behavior. In that case, use properties to hide functional implementation behind simple data attribute access syntax.

Note 1: Properties only work on new-style classes.

Note 2: Try to keep the functional behavior side-effect free, although side-effects such as caching are generally fine.

Note 3: Avoid using properties for computationally expensive operations; the attribute notation makes the caller believe that access is (relatively) cheap.

- If your class is intended to be subclassed, and you have attributes that you do not want subclasses to use, consider naming them with double leading underscores and no trailing underscores. This invokes Python's name mangling algorithm, where the name of the class is mangled into the attribute name. This helps avoid attribute name collisions should subclasses inadvertently contain attributes with the same name.

Note 1: Note that only the simple class name is used in the mangled name, so if a subclass chooses both the same class name and attribute name, you can still get name collisions.

Note 2: Name mangling can make certain uses, such as debugging and __getattr__(), less convenient. However the name mangling algorithm is well documented and easy to perform manually.

Note 3: Not everyone likes name mangling. Try to balance the need to avoid accidental name clashes with potential use by advanced callers.

Programming Recommendations

- Code should be written in a way that does not disadvantage other implementations of Python (PyPy, Jython, IronPython, Pyrex, Psyco, and such).

For example, do not rely on CPython's efficient implementation of in-place string concatenation for statements in the form a+=b or a=a+b. Those statements run more slowly in Jython. In performance sensitive parts of the library, the ''.join() form should be used instead. This will ensure that concatenation occurs in linear time across various implementations.

- Comparisons to singletons like None should always be done with 'is' or 'is not', never the equality operators.

Also, beware of writing "if x" when you really mean "if x is not None"

-- e.g. when testing whether a variable or argument that defaults to None was set to some other value. The other value might have a type (such as a container) that could be false in a boolean context!

- Use class-based exceptions.

String exceptions in new code are forbidden, because this language feature is being removed in Python 2.6.

Modules or packages should define their own domain-specific base exception class, which should be subclassed from the built-in Exception class. Always include a class docstring. E.g.:

```
class MessageError(Exception):
```

"""Base class for errors in the email package."""

Class naming conventions apply here, although you should add the suffix "Error" to your exception classes, if the exception is an error. Non-error exceptions need no special suffix.

- When raising an exception, use "raise ValueError('message')" instead of the older form "raise ValueError, 'message'".

The paren-using form is preferred because when the exception arguments are long or include string formatting, you don't need to use line continuation characters thanks to the containing parentheses. The older form will be removed in Python 3000.

- When catching exceptions, mention specific exceptions whenever possible instead of using a bare 'except:' clause.

For example, use:

```
try:
    import platform_specific_module
except ImportError:
    platform_specific_module = None
```

A bare 'except:' clause will catch SystemExit and KeyboardInterrupt exceptions, making it harder to interrupt a program with Control-C, and can disguise other problems. If you want to catch all exceptions that signal program errors, use 'except Exception:'.

A good rule of thumb is to limit use of bare 'except' clauses to two cases:

1) If the exception handler will be printing out or logging the traceback; at least the user will be aware that an error has occurred.

2) If the code needs to do some cleanup work, but then lets the exception propagate upwards with 'raise'. 'try...finally' is a better way to handle this case.

- Additionally, for all try/except clauses, limit the 'try' clause to the absolute minimum amount of code necessary. Again, this avoids masking bugs.

Yes:

```
try:
    value = collection[key]
except KeyError:
    return key_not_found(key)
else:
    return handle_value(value)
```

No:

```
try:
    # Too broad!
    return handle_value(collection[key])
except KeyError:
    # Will also catch KeyError raised by handle_value()
    return key_not_found(key)
```

- Use string methods instead of the string module.

String methods are always much faster and share the same API with unicode strings. Override this rule if backward compatibility with Pythons older than 2.0 is required.

- Use ''.startswith() and ''.endswith() instead of string slicing to check for prefixes or suffixes.

startswith() and endswith() are cleaner and less error prone. For example:

    Yes: if foo.startswith('bar'):

    No: if foo[:3] == 'bar':

The exception is if your code must work with Python 1.5.2 (but let's hope not!).

- Object type comparisons should always use isinstance() instead of comparing types directly.

    Yes: if isinstance(obj, int):

    No: if type(obj) is type(1):

When checking if an object is a string, keep in mind that it might be a unicode string too! In Python 2.3, str and unicode have a common base class, basestring, so you can do:

    if isinstance(obj, basestring):

In Python 2.2, the types module has the StringTypes type defined for that purpose, e.g.:

    from types import StringTypes

    if isinstance(obj, StringTypes):

In Python 2.0 and 2.1, you should do:

```
    from types import StringType, UnicodeType

    if isinstance(obj, StringType) or \

        isinstance(obj, UnicodeType) :
```

- For sequences, (strings, lists, tuples), use the fact that empty sequences are false.

```
    Yes: if not seq:

         if seq:

    No:  if len(seq)

         if not len(seq)
```

- Don't write string literals that rely on significant trailing whitespace. Such trailing whitespace is visually indistinguishable and some editors (or more recently, reindent.py) will trim them.

- Don't compare boolean values to True or False using ==

```
    Yes:   if greeting:

    No:    if greeting == True:

    Worse: if greeting is True:
```

References

[1] PEP 7, Style Guide for C Code, van Rossum

[2] http://www.python.org/doc/essays/styleguide.html

[3] PEP 257, Docstring Conventions, Goodger, van Rossum

[4] http://www.wikipedia.com/wiki/CamelCase

[5] Barry's GNU Mailman style guide http://barry.warsaw.us/software/STYLEGUIDE.txt

[6] PEP 20, The Zen of Python

[7] PEP 328, Imports: Multi-Line and Absolute/Relative

Copyright

Glosario

Ambientes Integrados de Desarrollo (IDE): aplicación que provee facilidades para el desarrollo de software, por parte de los programadores [Wikipedia, 2008k].

API: aplication programming interface, conjunto de funciones utilizado para comunicarse con otros programas [Wikipedia, 2008l].

buildroot: es la ubicación final del software instalado [Foster-Johnson, E., 2003].

Cabecera de la estructura: es una parte de los ficheros con formato .rpm que contendrá cero o más piezas de datos. [Bailey, E.C., 2000]

Checksum: suma de control, valor numérico utilizado para verificar la integridad de un bloque de datos. [Tejada Shoebridge S.L., 2002]

Código abierto: es una de las libertades de software libre, permite el acceso al código fuente de un programa y modificarlo a conveniencia.

Dependencias: software de los que depende el correcto funcionamiento de otro programa.

Estándares de UNIX: en el diseño y desarrollo de UNIX desde sus inicios se siguieron ciertos patrones, aunque en realidad no se define un estándar oficial hasta 1988 con el IEEE POSIX.

Fichero de configuración: fichero usado para guardar la configuración inicial de algunos programas de computadoras. [Wikipedia, 2008m]

Firma digital: método criptográfico que identifica un mensaje o documento. [Wikipedia, 2008n]

Gentoo Linux: sistema operativo libre basado en GNU/Linux y FreeBSD. [Gentoo Foundation, 2008a]

GNU: Es el nombre del proyecto creado en 1984 por Richard Stallman para crear un sistema operativo totalmente libre, es un acrónimo recursivo que significa “GNU no es Unix”. [Free Software Foundation,

2007]

GNU/Linux: sistema operativo desarrollado por la comunidad del Proyecto GNU con el núcleo Linux. [Free Software Foundation, 2008]

GTK+: grupo de librerias multiplataforma para desarrollar interfaces gráficas GUI (Graphic User Interface), desarrolladas por el Proyecto GNU. [Wikipedia, 2008o]

IEEE Posix Shell: estándar de la IEEE, que contiene normas para el uso e interacción con el shell de Unix.

Kernel: componente central de un sistema operativo, encargado de gestionar los recursos a través de llamadas al sistema, también se le llama núcleo [Gutierrez, I., 2001; Gray, B., 2008; Wikipedia, 2008p; Cabanes, N., 2008].

Lenguaje máquina: lenguaje original de la computadora, a través del cual esta es capaz de ejecutar interpretar los programas [Enciclopedia, 2008].

Lenguajes de alto nivel: lenguajes de programación con un alto grado de abstracción y proximidad al lenguaje de los humanos [Enciclopedia, 2008].

Licencia Pública General (GPL): licencia libre y copyleft del proyecto GNU para software y otros trabajos [Free Software Foundation, 2007].

Linux: núcleo de sistema operativo desarrollado en 1991 por Linus Torvalds.

MD5: algoritmo de reducción criptográfico de 128 bits altamente usado. [Wikipedia, 2008q]

NCURSES: es una biblioteca de programación que permite al programador escribir interfaces basadas en texto, TUI (Text User Interface), desarrollada por el Proyecto GNU [Free Software Foundation, 2007c].

network file system (NFS): sistema de ficheros distribuido [Siever, E., Weber, A., 2005; Smith, C., 2006].

Parches: patch, es un fragmento de código que se agrega a un software para corregir algún error dentro

del mismo. [Avizora, 2008]

PM: gestor de paquetes diseñado por Rik Faith y Doug Hoffman para Red Hat, utilizando las ventajas y desventajas de los anteriores. [Bailey, E.C., 2000]

PMS: gestor de paquetes desarrollado por un equipo liderado por Rik Faith, usado por la distribución BOGUS. [Bailey, E.C., 2000]

Portage: herramienta de gestión de software de Gentoo Linux. [Gentoo Foundation, 2008b]

RPP: primer gestor de paquetes de la distribución Red Hat. [Bailey, E.C., 2000]

Script: un programa que interactúa con una aplicación o con otro programa utilitario. [Microsoft Corporation, 2002]

Tiempo de compilación: tiempo que demora el sistema en traducir un código escrito un lenguaje de programación a lenguaje máquina.

UNIX: sistema operativo desarrollado en 1969 por un equipo de los laboratorios AT&T. [The Open Group, 2007]

wxPython: es una adaptación de la biblioteca gráfica wxWidgets para trabajar con el lenguaje de programación Python [wxPython, 2008].

XML: metalenguaje para definir documentos con formatos y estándares comunes, diseñado por la W3C [Definición, 2008b].

XML-RPC: conjunto de aplicaciones de software que que permite la comunicación entre sistemas a través del lenguaje XML y el protocolo http [UserLand Software, 2008].