Universidad de las Ciencias Informáticas

Núcleo del entorno de escritorio de Nova Ligero 2015

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

Autores:
Yaritza Yanet González Silva
Alexis López Zubieta

Tutores:
MsC. Allan Pierra Fuentes
Ing. Dairelys García Rivas

La Habana
Junio de 2014

DECLARACIÓN DE AUTORIA

Declaramos ser los únicos autores de este trabajo y autorizamos al centro CESOL de la Universidad de las Ciencias Informáticas; para que haga el uso que estimen pertinente con este trabajo.

Para que así conste firmamos la presente a los ____ días del mes de _____ del año____.

Yaritza Yanet González Silva

_____

Firma del autor

Alexis López Zubieta

_____

Firma del autor

MsC. Allan Pierra Fuentes

_____

Firma del tutor

Ing. Dairelys García Rivas

_____

Firma del tutor

“Sé el cambio que quieres ver en el mundo”

Mahatma Gandhi

AGRADECIMIENTOS

…. a Fidel y la Revolución por darme la oportunidad de superarme.

… a mi mamita bella por su esfuerzo y dedicación.

... a mi papi por ser mi guía en los momentos difíciles.

…. A mi novio por su apoyo y compañía.

... a toda mi familia.

…. a los tutores Allan y Dairelys por su apoyo.

Yaritza Yanet González Silva

A todos aquellos que de una forma u otra

han contribuido a mi formación como ingeniero.

En especial a mi novia y a mi compañera de tesis.

Alexis López Zubieta

DEDICATORIA

A toda mi familia y a todos los que de una manera u otra me han apoyado.

Yaritza Yanet González Silva

A toda mi familia en especial a mis padres y a mi hermano

Alexis López Zubieta

Resumen

El entorno de escritorio de la distribución de software Nova Ligero, se encuentra orientado a equipos de bajas prestaciones de hardware. Este es producto de una bifurcación del proyecto LXDE, desarrollado por aficionados al software libre. En consecuencia hereda un conjunto de deficiencias causadas por la no aplicación de un proceso de desarrollo de software formal. Específicamente el producto posee una pobre interoperabilidad entre los componentes que lo integran. El objetivo de la presente investigación es el desarrollo del Núcleo del entorno de escritorio de Nova Ligero 2015, para permitir la interoperabilidad entre sus componentes sin afectar la disponibilidad de recursos. Los principales aportes de esta investigación son: una aplicación encargada de integrar componentes desarrollados como bibliotecas compartidas y permitir la interoperabilidad entre los mismos; y el diseño de una propuesta de arquitectura para el entorno de escritorio de Nova Ligero 2015.

Palabras clave: disponibilidad de recursos, entorno de escritorio, interoperabilidad, Nova Ligero.

Índice

Introducción....1
Capítulo 1: “CONCEPTOS FUNDAMENTALES Y ESTADO DEL ARTE DE LAS TECNOLOGÍAS”....5
1.1 Conceptos asociados al dominio del problema....5
1.1.1 Entorno de escritorio....5
1.1.2 Interoperabilidad entre aplicaciones de escritorio....6
1.2 Arquitectura de Guano....6
1.3 Interoperablilidad....7
1.4.2 Estrategia....8
1.4.2.1 Arquitectura Basada en Componentes....8
1.4.2.2 Arquitectura Orientada a Servicios....9
1.4.2.2 Mecanismos de comunicación....11
1.5 Sistemas Homólogos....13
1.5.1 Sistema de Plug-ins de KDE....13
1.5.2 Sistema de Plug-ins de Leechcraft....14
1.5.3 Marco de trabajo OSGi....14
1.5.4 Sistema de Plug-ins de Eclipse....15
1.5.5 CTK Plugin Framework....15
1.5.6 Biblioteca CppMicroServices....15
1.5.7 Comparación de soluciones....16
1.6 Metodología a utilizar....16
1.7 Herramientas a utilizar....18
1.8 Lenguajes a utilizar....19
1.7 Conclusiones del capítulo....20
Capítulo 2: “DISEÑO E IMPLEMENTACIÓN DE LA SOLUCIÓN PROPUESTA”....21
2.1 Propuesta de solución....21
2.2 Modelo de Dominio....21
2.2.1 Diagrama de clases del Modelo de Dominio....22
2.2.2 Descripción de las clases del Modelo de Dominio....22
2.3 Modelado del Sistema....23
2.3.1 Requisitos del sistema....23
2.3.1.1 Requisitos funcionales....23
2.3.1.2 Requisitos no funcionales....24
2.3.2 Vista de Casos de Uso....24
2.3.2.1 Actores del Sistema....25
2.3.2.2 Diagrama de Casos de Uso del Sistema....25
2.3.2.3 Descripción de los Casos de Uso....26
2.4 Propuesta de la Arquitectura del sistema....34
2.5 Vista Lógica....38
2.5.1 Patrones de diseño....38
2.5.2 Descripción de las clases del sistema....39
2.5.3 Estándares de codificación....40
2.6 Vista de Desarrollo....42
2.6.1 Diagrama de componentes....42
2.7 Conclusiones del capítulo....42
Capítulo 3: “VALIDACIÓN DE LA SOLUCIÓN PROPUESTA”....43

3.1 Perfil de configuración para las pruebas....43
3.2 Validación de las interfaces del Núcleo....44
3.2.1 Interfaz IController....44
3.2.2 Interfaz IModuleManager....45
3.2.3 Interfaz IEnvironment....46
3.2.4 Interfaz ISettingsProfile....47
3.3 Validación de los requisitos relacionados al desarrollo de componentes....48
3.4 Validación de los requisitos no funcionales....49
3.4.1 Tiempo de inicio....50
3.4.2 Consumo de memoria....50
3.5 Análisis del impacto de la propuesta de arquitectura en el consumo de recursos del EE....51
3.6 Conclusiones del Capítulo....53
Conclusiones Generales....54
Recomendaciones....55
Referencias Bibliográficas....56
Bibliografía Consultada....61
Glosario de Términos....62
Anexos....63
Anexo 1: Descripción del caso de uso Implementar componente....63
Anexo 2: Descripción del caso de uso Modificar componente....63
Anexo 3: Descripción del caso de uso Desplegar componente....63
Anexo 4: Descripción del caso de uso Configurar perfil....64
Anexo 5: Descripción del caso de uso Gestionar variable de entorno....64
Anexo 6: Plantilla para el desarrollo de componentes....66

Introducción

Una distribución GNU/Linux es una distribución de software basada en el núcleo Linux que incluye determinados paquetes de software para satisfacer las necesidades de un grupo específico de usuarios (Koch, 2005). Actualmente estas distribuciones se han vuelto cada vez más populares entre los usuarios comunes, principalmente por la estabilidad y seguridad que ofrecen; dentro de las más destacadas se encuentran: Debian, Ubuntu, Archlinux, Gentoo, Fedora y OpenSuse (YERPES, 2012).

En el país con el objetivo de satisfacer la necesidad de migración a plataformas de código abierto, para el logro de la soberanía tecnológica, se crea la distribución GNU/Linux Nova en la Facultad 10 de la Universidad de las Ciencias Informáticas. Para que el proceso de migración fuera efectivo en todos los sistemas, se crearon tres variantes: Nova Servidores, Nova Escritorio y Nova Ligero. Esta última surge por la necesidad de aumentar la vida útil de los dispositivos que quedan obsoletos por la rápida evolución de los sistemas informáticos (Pierra, 2011). Nova ligero fue conformado mediante la reutilización de software existente en los repositorios de Ubuntu, Debian y Fedora, siguiendo el proceso definido por la "Metodología Nova OpenUp para el desarrollo de distribuciones GNU/Linux" (Fernández, 2012). Los productos de software incluidos, además de satisfacer los requerimientos de los involucrados[1], deben consumir la menor cantidad posible de RAM[2] y ciclos de CPU[3], de manera que estos queden disponibles para aplicaciones que requieren un mayor consumo. Por esta razón se rechazan productos como GNOME[4] y KDE[5], en su lugar se utiliza el producto LXDE[6] (Rodríguez, 2009). Este brinda un Entorno de Escritorio (EE[7]) funcional para los usuarios con conocimientos avanzados de informática (LXDE.ORG, 2012). Con el objetivo de acercar LXDE a los usuarios comunes, el proyecto Nova bifurca dicho producto y se crea Guano, entorno de escritorio ligero conformado por aplicaciones heredadas del proyecto LXDE (Albalat, 2009). Durante este período fue objeto en varias ocasiones de mantenimiento correctivo[8], siendo validado en 2012 por la empresa certificadora de calidad de software CALISOFT (CALISOFT, 2009) (García, 2012). En las diferentes iteraciones del proceso de mantenimiento de Guano se detectaron varias deficiencias que no constituyen problemas funcionales pero que afectan el desempeño del sistema. Estas

1 Quienes pueden afectar o son afectados en el proceso de desarrollo y mantenimiento de un producto de software.
2 **RAM:** Memoria de Acceso Aleatorio por sus siglas en Inglés.
3 **CPU:** La unidad central de procesamiento, CPU (por sus siglas del inglés Central Processing Unit), o, simplemente, el procesador, es el componente en una computadora digital que interpreta las instrucciones y procesa los datos contenidos en los programas de computadora. **Ciclos de CPU:** Operaciones matemáticas o lógicas que realiza la Unidad Central de Procesamiento.
4 Entorno de escritorio desarrollado por la comunidad GNOME.
5 Entorno de escritorio desarrollado por la comunidad KDE.
6 Entorno de escritorio ligero desarrollado por la comunidad LXDE.
7 Conjunto de software para ofrecer al usuario de una computadora una interacción amigable y cómoda.
8 Proceso de mantenimiento orientado a corregir deficiencias de un producto de software.

no se localizan en un solo componente, más bien afectan al EE en su conjunto. A continuación se exponen algunas de las deficiencias detectadas:

- Las herramientas de configuración de intermediarios (Proxy) e idioma utilizan las variables de entorno del sistema como medio para compartir datos, procedimiento que se considera inapropiado porque requiere el reinicio de sesión para hacer efectivas las configuraciones.
- Por otra parte el panel, el gestor de ventanas, el gestor de ficheros y el centro de configuraciones permiten la selección y ejecución de aplicaciones utilizando mecanismos diferentes para obtener el nombre, iconos y ejecutables de las mismas. Estos mecanismos al ser diferentes duplican las funcionalidades en cuestión así como los datos asociados a las aplicaciones.
- La ausencia de un único sistema de gestión de atajos de teclado constituye otra deficiencia presente en el EE de Nova Ligero. Las aplicaciones que lo integran implementan esta funcionalidad de manera independiente propiciando colisiones cuando un atajo es definido para realizar diferentes acciones. De manera similar a la situación anterior se incurre en la duplicación de la implementación de la funcionalidad y los datos asociados.
- Por último el EE posee 22 aplicaciones que son cargadas una vez iniciada una sesión, estas fueron desarrolladas utilizando el paquete de herramientas gráficas (GTK). Cada aplicación desarrollada con estas herramientas consume aproximadamente 1 MiB por concepto del ciclo de ejecución de GTK[9]. Teniendo en cuenta que esta condición se cumple para cada una de las aplicaciones, se afirma que el EE invierte aproximadamente 22 MiB.

Deficiencias similares a las descritas anteriormente se repiten en las demás aplicaciones del EE. Como posible solución se puede considerar la modificación de las mismas, de manera que compartan las funcionalidades y datos necesarios, interoperen[10], mediante mecanismos de comunicación entre procesos (IPC)(Mitchell, Oldham, Samuel, 2001).

Los mecanismos de comunicación permiten el intercambio de datos y funcionalidades entre aplicaciones desarrolladas con tecnologías diferentes gracias a la transformación de estos antes de ser transmitidos y al ser recibidos. En el proceso se añade una capa de abstracción[11] adicional y se duplican los datos, lo cual afecta negativamente la disponibilidad de recursos[12] del sistema.

---

[9] **Ciclo de ejecución de GTK**: conjunto de funcionalidades de GTK destinadas a gestionar los eventos y respuestas de una aplicación.

[10] **Interoperabilidad:** es la habilidad de dos o más componentes de software de intercambiar información y usar la información intercambiada.

[11] **Capa de abstracción:** procedimiento para ocultar los detalles de implementación de un conjunto de funcionalidades.

[12] **Recursos:** Memoria de Acceso Aleatorio (RAM), capacidad de almacenamiento y Unidad Central de Procesamiento (CPU) con que cuenta un computador.

Por otra parte los componentes del EE fueron desarrollados de manera monolítica lo que implica que sus módulos, clases y funciones tienen fuertes dependencias entre sí. La modificación de una sección del programa provocaría inconsistencias en el resto del mismo y por consiguiente será necesario la modificación en cadena del resto de las secciones que dependan de esta. Esta situación dificulta su adaptación a contextos diferentes y atenta contra la mantenibilidad de la solución a largo plazo.

Por ello se afirma que el uso de mecanismos IPC no constituye una solución viable para lograr la interoperabilidad entre los componentes del EE, mas este problema se remite a deficiencias en el diseño de los componentes.

Atendiendo a lo anteriormente expuesto se plantea la siguiente **situación problemática**: los componentes que conforman el EE de Nova Ligero no están diseñados para interoperar, lo cual afecta la disponibilidad de recursos. Formulándose el **problema científico** de la siguiente manera: ¿Cómo permitir la interoperabilidad entre componentes del entorno de escritorio de Nova Ligero 2015 sin afectar la disponibilidad de recursos?

Por tanto el **objeto de estudio** lo constituye: la interoperabilidad entre componentes de entornos de escritorios ligeros y el **campo de acción**: la interoperabilidad entre componentes del entorno de escritorio de Nova Ligero 2015. Para brindarle una solución efectiva al problema, se plantea como **objetivo general:** Desarrollar el Núcleo del entorno de escritorio de Nova Ligero 2015, para permitir la interoperabilidad entre sus componentes sin afectar la disponibilidad de recursos.

Para alcanzar este objetivo general se plantean los siguientes **objetivos específicos:**

1. Identificar las tendencias de investigación en el campo de la interoperabilidad de sistemas de software prestando atención a las aplicaciones de escritorio a través de la revisión de la bibliografía disponible.
2. Diseñar e implementar el Núcleo del entorno de escritorio de Nova Ligero 2015.
3. Validar el Núcleo del entorno de escritorio de Nova Ligero 2015 mediante la realización de pruebas unitarias.

Luego de realizada una revisión de la literatura, se propone la siguiente **idea a defender:** El desarrollo del Núcleo del entorno de escritorio de Nova Ligero 2015, permitirá la interoperabilidad entre sus componentes sin afectar la disponibilidad de recursos.

Para el desarrollo de la investigación se plantean las siguientes **tareas de investigación:**

1. Identificación de ventajas y desventajas de los enfoques de interoperabilidad entre sistemas de software.

2. Evaluación del contenido de la información obtenida sobre la interoperabilidad entre sistemas de software.
3. Identificación de patrones de diseño aplicables en el desarrollo de la solución.
4. Implementación del Núcleo del entorno de escritorio de Nova Ligero 2015.
5. Planificación y ejecución de pruebas unitarias a la solución.

Para dar cumplimiento al objetivo general y respuesta al problema planteado, explorar y procesar la información referente al objeto de estudio de la investigación, se utilizaron diferentes métodos teóricos y empíricos:

Métodos teóricos: como métodos teóricos se identificaron el histórico – lógico para el estudio de trabajos anteriores relacionados con el problema planteado y el analítico – sintético para dividir el problema en subproblemas que faciliten su estudio y luego sintetizarlo en una solución general acorde al problema propuesto.

Métodos empíricos: como métodos empíricos se identificó el análisis – documental para la revisión de la literatura necesaria durante el proceso de investigación.

Estructura del documento

Capítulo 1: "CONCEPTOS FUNDAMENTALES Y ESTADO DEL ARTE DE LAS TECNOLOGÍAS": Este capítulo está compuesto por los conceptos fundamentales utilizados durante la investigación, los tipos de arquitecturas relevantes al logro de la propuesta realizada, un estudio de los distintos sistemas desarrollados sobre la base de la solución de esta investigación y las herramientas, lenguajes y metodología a utilizar durante el desarrollo de la solución propuesta.

Capítulo 2: "DISEÑO E IMPLEMENTACIÓN DE LA SOLUCIÓN PROPUESTA": Este capítulo está compuesto por los artefactos generados durante el modelado del sistema, estos son: modelo de dominio, modelo de casos de uso, de clases del diseño, y de componentes. Se exponen además el estilo arquitectónico, los principios de diseño, patrones de diseño y estándares de codificación utilizados durante el desarrollo de la solución propuesta.

Capítulo 3: "VALIDACIÓN DE LA SOLUCIÓN PROPUESTA": Este capítulo está compuesto por las pruebas realizadas al software, relacionadas con las funcionalidades que este ofrece, su consumo de memoria y tiempo de respuesta. Por último se analiza el impacto de la arquitectura propuesta en la disponibilidad de recursos del sistema.

Capítulo 1: “CONCEPTOS FUNDAMENTALES Y ESTADO DEL ARTE DE LAS TECNOLOGÍAS”

El desarrollo constante de disímiles sistemas informáticos, trajo consigo la necesidad de comunicarlos, ya sea para aumentar el conjunto de funcionalidades a ofrecer o minimizar el esfuerzo de desarrollo. Para establecer la comunicación entre los distintos sistemas, existen estilos arquitectónicos y mecanismos de comunicación que facilitan el proceso de integración. En este capítulo se estarán abordando los conceptos asociados al dominio del problema, los aspectos importantes a tener en cuenta como parte del resultado de esta investigación, y las herramientas, lenguajes y metodología a utilizar durante el desarrollo de la solución propuesta.

1.1 Conceptos asociados al dominio del problema

1.1.1 Entorno de escritorio

El término entorno de escritorio comprende un conjunto de aplicaciones que combinadas ofrecen al usuario del ordenador una interfaz gráfica similar a un escritorio tradicional (KAPTELININ, 2007). Esta metáfora se compone a partir de un espacio bidimensional donde se colocan archivos y carpetas, como estructuras para el almacenamiento de datos, y ventanas para el manejo de herramientas como calculadoras, editores de texto, etc. (KAPTELININ, 2007). Los entornos de escritorio contemporáneos suelen extender esta interfaz con otros componentes como paneles, menús, áreas de notificaciones y otros que a pesar de no estar comprendidos dentro del concepto original de un escritorio aportan nuevas funcionalidades al sistema.

El entorno de escritorio constituye, para el usuario, un medio de acceso a las diferentes funcionalidades y datos en el ordenador. Permite la gestión de aplicaciones y su integración con el resto del sistema y utiliza componentes de hardware, como dispositivos de almacenamiento externos, periféricos, dispositivos de interconexión y otros desde el espacio de usuario[13].

Atendiendo al consumo de RAM los entornos de escritorio, suelen dividirse en dos categorías: pesados y ligeros. Entre los pesados se encuentran K Desktop Environment (KDE), Gnome, Cinamon y Deepin. Estos tienen un consumo de memoria superior a los 50Mb y cuentan con abundantes efectos visuales como animaciones, transparencias y otros que requieren de una mayor capacidad de procesamiento (LARABEL, 2010). Por otro lado, los EE ligeros eliminan estos efectos y limitan sus funcionalidades en aras de lograr un mejor rendimiento. Entre los principales entornos ligeros se encuentran XFCE, LXDE,

[13] **Espacio de usuario:** espacio de aplicación externo al núcleo (aplicaciones utilizadas por el usuario final).

Enlightment y otros que no proveen todos los componentes de la metáfora de escritorio como IceWin, WindowMaker, Openbox y FluxBox.

1.1.2 Interoperabilidad entre aplicaciones de escritorio

Tradicionalmente los sistemas GNU/Linux se han compuesto por un conjunto de herramientas que a pesar de ser diseñadas para una funcionalidad específica pueden ser integradas para cumplir tareas no concebidas por sus creadores. Esto le otorga una gran adaptabilidad al sistema en general. Con la aparición de las interfaces gráficas de usuario y la utilización de nuevos dispositivos de entrada como el ratón y las pantallas táctiles cambia la manera en que el usuario se comunica con el ordenador. En consecuencia, aumentó la necesidad de interacción entre las aplicaciones del sistema.

Las aplicaciones de escritorio contemporáneas no solo tienen que procesar un conjunto de datos de entrada y generar una salida, mas, deben responder ante los eventos del sistema y adaptar su comportamiento (SWEET, 2001). Este tipo de situación se evidencia en los siguientes ejemplos:

- El gestor de ventanas debe ofrecer al usuario una retroalimentación del estado de las ventanas.
- Las aplicaciones deben ser capaces de encontrar y utilizar funcionalidades provistas por otras, por ejemplo, el navegador web debe ser capaz de utilizar la funcionalidad “enviar correo” implementada en algún cliente de correo.
- Las aplicaciones deben estar al tanto de los cambios en la configuración del sistema, cambio de tema de iconos, estilo de texto, paleta de colores, utilización de un intermediario para la conexión a internet, conexión de dispositivos de hardware y otras.
- Los eventos relacionados a las diferentes tareas que ejecuta el usuario deben ser notificados al mismo.

Tanto EE pesados como ligeros deben ser capaces de interoperar con el resto de las aplicaciones del sistema para lograr una mejor experiencia de usuario y evitar la duplicidad de datos y funcionalidades.

1.2 Arquitectura de Guano

En la etapa de diseño se define la arquitectura del software, esta refleja las características del mismo y constituye un paso crítico en el proceso de desarrollo. Guano al estar compuesto por aplicaciones del EE LXDE hereda su arquitectura. Este producto es desarrollado por una comunidad de desarrolladores aficionados al software libre que ignora los procedimientos formales de desarrollo de software. Como resultado, la arquitectura de LXDE no se encuentra bien definida y por transitividad la de Guano. Resulta

costoso realizarle el mantenimiento, ya que no existe la documentación necesaria mediante la cual entender el software para posteriormente someterlo a cambios.

La arquitectura de Guano se asemeja a una Arquitectura Basada en Componentes porque se constituye a partir de aplicaciones independientes, pero estas carecen de interfaces de programación de aplicaciones (API[14], *Application Programming Interface*), las cuales son necesarias para el intercambio de datos y funcionalidades. Esto es conocido en la literatura como interoperabilidad.

1.3 Interoperablilidad

El Instituto de Ingenieros Eléctricos y Electrónicos (IEEE) en el Glosario Estándar de Terminología de Ingeniería de Software (IEEE, 1990) define **interoperabilidad** como:

> "La habilidad de dos o más sistemas o componentes de intercambiar información y usar la información intercambiada."

En tanto el Comité Técnico de la Organización Internacional de Estandarización (ISO) en la norma ISO/DIS 19101 (ISO, 2001) brinda la siguiente definición:

> "Capacidad de un sistema o componente de un sistema para proporcionar intercambio de información y el procesamiento cooperativo entre aplicaciones."

A partir de las definiciones analizadas y teniendo en cuenta la bibliografía consultada, la presente investigación determina que el desarrollo de un EE interoperable permitirá (MENECES, 2010):

- La integración de nuevos componentes en el futuro.
- Lograr mayor cooperación entre componentes.
- La disponibilidad y visibilidad de los datos entre componentes.
- Eliminar la duplicidad de datos y funcionalidades.

1.4.2 Estrategia

Esta investigación identifica como estrategia para lograr la interoperabilidad en un sistema de software el uso de estilos arquitectónicos basados en componentes, con el objetivo de facilitar el desarrollo y mantenimiento del sistema. Este podrá ser dividido en componentes simples que compongan

[14] **API:** Interfaz de programación de aplicaciones: es el conjunto de funciones y procedimientos (o métodos, en la programación orientada a objetos) que ofrece cierta biblioteca o componente para ser utilizado por otro software como una capa de abstracción. Son usadas generalmente en las bibliotecas.

componentes más complejos y que respondan a una única funcionalidad. A continuación se analizan las características de los posibles estilos arquitectónicos a utilizar:

1.4.2.1 Arquitectura Basada en Componentes

La Arquitectura Basada en Componentes (ABC) consiste en la descomposición del software en componentes funcionales con interfaces bien definidas que permitan la comunicación entre componentes (Meier, 2009). Tiene asociada una serie de beneficios, los cuales han sido considerados por la Ingeniería de Software, creando su propio espacio (Ingeniería de Software Basada en Componentes) que guíe el desarrollo basado en componentes.

Entiéndase por **componente de software** como:

- "Parte no trivial, casi independiente y reemplazable de un sistema que cumple una función dentro del contexto de una arquitectura y libera un conjunto de servicios que son usados solosólo a través de interfaces bien definidas (Brown, 1998)."

Principios Fundamentales de la ABC:

Los componentes de una ABC deben cumplir con los siguientes principios (SAN, 2013):

- **Reusabilidad:** Los componentes deben ser diseñados para ser utilizados en escenarios diferentes por diferentes aplicaciones, sin embargo, algunos pueden ser diseñados para tareas específicas.
- **Sin contexto específico:** Los componentes deben ser diseñados para operar en diferentes ambientes y contextos.
- **Extensibilidad:** Un componente debe poder ser extendido a partir de otros componentes para crear un nuevo comportamiento.
- **Encapsulamiento:** Los componentes deben exponer interfaces que permitan a otros componentes o aplicaciones usar sus funcionalidades sin revelar detalles internos de los procesos que realizan o de su estado.
- **Independiente:** Los componentes deben estar diseñados para tener dependencias mínimas con otros componentes, con el objetivo que puedan ser instalados en el ambiente adecuado sin afectar otros componentes o sistemas.

Beneficios del uso de una ABC:

- Permite la evolución de los componentes sin afectar a los demás o al sistema como un todo.

- Permite la reutilización de componentes desarrollados por terceros, lo cual minimiza el costo de desarrollo y mantenimiento.
- Reusabilidad de los componentes, por su independencia de contextos y la posibilidad de ser utilizados en otros sistemas.

1.4.2.2 Arquitectura Orientada a Servicios

La Arquitectura Orientada a Servicios (SOA, en inglés *Service Oriented Architecture*), es definida como:

- "Estilo resultante de políticas, prácticas y frameworks (marcos de trabajo) que permiten que la funcionalidad de una aplicación se pueda proveer y consumir como conjuntos de servicios (Mangarelli, 2006)."

Esta permite la creación de sistemas altamente flexibles, a su vez brinda una forma bien definida de exposición e invocación de servicios, facilitando la interacción entre diferentes sistemas propios o de terceros. Además facilita la integración con sistemas legados, reduciendo los costos de implementación (KOSKELA, 2007).

Esta arquitectura define los siguientes términos:

Término	**Definición / Comentario**
Servicio	Una función sin estado, que acepta una(s) llamada(s) y devuelve una(s) respuesta(s) mediante una interfaz bien definida.
Sin estado	No mantiene ni depende de condición pre-existente alguna. Los servicios no son dependientes de la condición de ningún otro servicio. Reciben en la llamada toda la información que necesitan para dar una respuesta.
Proveedor	La función que brinda un servicio en respuesta a una llamada o petición desde un consumidor.
Consumidor	La función que consume el resultado del servicio provisto por un proveedor.

Tabla 1: Términos de una SOA

SOA define las siguientes capas de software:

- **Aplicaciones básicas:** Sistemas desarrollados bajo cualquier arquitectura o tecnología, geográficamente dispersos.
- **Exposición de funcionalidades:** Donde las funcionalidades de la capa aplicativa son expuestas en forma de servicios.

- **Integración de servicios:** Facilitan el intercambio de datos entre elementos de la capa aplicativa orientada a procesos internos o en colaboración.
- **Entrega:** Donde los servicios son desplegados a los usuarios finales.

Principios de una SOA:

- **Los Servicios deben ser reusables:** Todo servicio debe ser diseñado y construido pensando en su reutilización dentro del dominio público para su uso masivo.
- **Los Servicios deben proporcionar un contrato formal:** Todo servicio desarrollado, debe proporcionar un contrato en el cual figuren: el nombre del servicio, su forma de acceso, las funcionalidades que ofrece, los datos de entrada de cada una de las funcionalidades y los datos de salida. De esta manera, todo consumidor del servicio, accederá a este mediante el contrato, logrando así la independencia entre el consumidor y la implementación del propio servicio.
- **Los Servicios deben poseer bajo acoplamiento:** Los servicios tienen que ser independientes los unos de los otros. Para lograr el bajo acoplamiento, cuando se ejecute un servicio, se accederá a él a través del contrato, logrando así la independencia entre el servicio que se va a ejecutar y el que lo llama. Si se consigue bajo acoplamiento, entonces los servicios podrán ser totalmente reutilizables.

Beneficios del uso de una SOA:

- Permite la reutilización e integración de productos desarrollados por terceros.
- Proporciona flexibilidad al sistema.
- Permite reemplazar elementos de la capa aplicativa SOA sin irrupción en el proceso de negocio.
- Facilita la integración de tecnologías disímiles.

Luego de analizados los estilos arquitectónicos compatibles con las necesidades del producto se decide utilizar el estilo orientado a servicios. Para ello se tuvo en cuenta que:

- El estilo ABC no comprende la gestión de dependencias en tiempo de ejecución. Por ello es imposible determinar cuando dos componentes están intercambiando datos y funcionalidades. En esta situación, si el componente proveedor es desactivado, el consumidor fallará junto con el sistema, afectándose negativamente la estabilidad del mismo. De manera alternativa se pueden cargar todos los componentes al inicio del sistema y mantenerse activos hasta el cierre pero se estaría gastando memoria en los componentes que no sean utilizados. Por último se afecta la flexibilidad del sistema, puesto que una vez definidos los componentes con sus relaciones no

pueden ser sustituidos por otros, lo cual impide que el sistema se adapte a las necesidades del usuario en tiempo de ejecución. Mientras que:

- El estilo SOA permite la gestión de dependencias, mediante el registro de contratos, donde se especifica el componente proveedor y consumidor del servicio. De esta manera se logra impedir la desactivación de componentes cuando sus servicios están siendo consumidos para evitar fallas en el sistema, o se permite que sean desactivados cuando no están siendo utilizados para ahorrar recursos. Este estilo obliga a que las relaciones entre los componentes se realice mediante servicios y no directamente, proporcionando flexibilidad al sistema. Los componentes solo deben conocer los servicios que van a consumir, no su proveedor, de manera que se pueda adicionar o eliminar componentes del sistema sin afectar los existentes.

1.4.2.2 Mecanismos de comunicación

Para establecer la comunicación entre componentes de una Arquitectura Orientada a Servicios, según el estudio realizado tradicionalmente se hace uso de los mecanismos de comunicación entre procesos (IPC[15], del inglés ***I**nter-**P**rocess **C**ommunication)*. A continuación se exponen los más destacados en sistemas GNU/Linux:

- **Memoria compartida:** consiste en establecer una región de memoria común para los procesos cooperativos[16]. De este modo los procesos pueden compartir información leyendo y escribiendo datos en la zona compartida. El acceso a esta región de memoria es tan rápido como el acceso a la memoria interna de los procesos (Mitchell, Oldham, Samuel, 2001). Este mecanismo es soportado por el núcleo en sistemas GNU/Linux.
- **Señales del sistema:** es una notificación asíncrona enviada a un proceso para informarle de un evento. Cuando se le manda una señal a un proceso, el sistema operativo modifica su ejecución normal. Si se había establecido anteriormente un procedimiento (*handler*) para tratar esa señal se ejecuta éste, si no se estableció nada previamente se ejecuta la acción por defecto para esa señal (Mitchell, Oldham, Samuel 2001). Este mecanismo es soportado por el núcleo en sistemas GNU/Linux .

[15] IPC: es un mecanismo que permite el intercambio de datos entre procesos. Los procesos pueden estar ejecutándose en uno o varios ordenadores conectados por una red. IPC ayuda a los programadores a organizar las actividades entre los diferentes procesos al proporcionar un conjunto de interfaces de programación.

[16] **Procesos cooperativos:** procesos que intercambiar datos y funcionalidades.

- **Pipes o Tuberías:** mecanismo de comunicación unidireccional que permite conectar automáticamente la salida estándar[17] de un programa con la entrada estándar[18] de otro programa (Jalón, 2000). Para mejorar el rendimiento, las tuberías se implementan utilizando espacios en memoria (buffers). Lo que permite al proceso proveedor, generar más datos que los que el proceso consumidor puede atender inmediatamente (Mitchell, Oldham, Samuel 2001). Este mecanismo es soportado por el núcleo en sistemas GNU/Linux.

- **Sockets:** representan un extremo de una comunicación bidireccional y tienen asociada una dirección IP, un protocolo y un número de puerto. Su función principal es permitir a dos programas (pueden estar situados en computadoras distintas) intercambiar cualquier flujo de datos, generalmente de manera fiable y ordenada. Su característica más importante es permitir la implementación de una arquitectura cliente-servidor, pues la comunicación es iniciada por uno de los programas que se denomina programa cliente y el segundo programa espera a que otro inicie la comunicación, denominándose programa servidor (Marquez, 2004)(Mitchell, Oldham, Samuel 2001). Este mecanismo es soportado por el núcleo en sistemas GNU/Linux.

- **Llamada a Procedimiento Remoto** (RPC**,** en inglés *Remote Procedure Call*)**:** protocolo de comunicación que permite a una aplicación ejecutar funciones declaradas en otra y obtener el resultado sin tener que manejar los detalles de la comunicación. Son utilizadas dentro del paradigma cliente-servidor. Constituye la evolución del mecanismo basado en Sockets (Marshall, 1999). Existen varias implementaciones de este mecanismo en sistemas GNU/Linux utilizando diferentes lenguajes de programación como: XML-RPC[19], DCE-RPC[20] y otros.

- **D-Bus:** es un mecanismo de IPC del tipo RPC, este constituye un estándar en la integración de aplicaciones de un mismo computador. Permite establecer la comunicación de forma sencilla entre los diferentes procesos que se ejecutan en una misma sesión de escritorio, asegurando una interoperabilidad óptima y minimizando el esfuerzo de integración.

  Los dos principales puntos de uso de D-Bus son los siguientes:

[17] **Salida estándar:** determina el destino del resultado de un programa o proceso.
[18] **Entrada estándar:** determina la fuente de datos de un programa o proceso.
[19] **XML-RPC:** es un protocolo de llamada a procedimiento remoto que usa XML para codificar los datos.
[20] **DCE-RPC:** es el sistema de llamada a procedimiento remoto desarrollado para el entorno de la informática distribuida.

- Comunicación entre aplicaciones de escritorio en la misma sesión, facilitando la integración de aplicaciones dentro de un mismo EE y el tratamiento de asuntos relativos al ciclo de vida de procesos.
- Comunicación entre el sistema operativo y la sesión de escritorio, incluyendo dentro del sistema operativo al núcleo y algunos demonios o procesos.

Los componentes del EE son aplicaciones independientes, por tanto requieren el uso de mecanismos IPC para interoperar. El uso de estos propicia la duplicidad de datos porque la información a compartir es duplicada al ser transformada a formato binario (proceso conocido como serialización[21]) y al ser convertida de formato binario a código de programación (deserialización[22]) es duplicada nuevamente. Este proceso debe realizarse siempre para que los datos de un proceso sean comprendidos por otro.

Esta investigación propone como alternativa al uso de los mecanismos IPC, integrar los componentes del EE en una única aplicación como bibliotecas dinámicas[23] (plug-in), con el objetivo de permitir la comunicación entre los componentes de manera directa, utilizando las técnicas propias del lenguaje de programación y no incurrir en la duplicación de datos compartidos.

1.5 Sistemas Homólogos

El estilo arquitectónico define de manera abstracta el modelo de comunicación entre componentes de un sistema (Reynoso, 2004). Esto se traduce primeramente en requisitos del sistema, luego en código de la aplicación. De esta manera aplicaciones con arquitecturas similares tienen secciones de código equivalentes. Por ello se analiza a continuación un conjunto de productos de software que siguen los estilos SOA o ABC, con el objetivo de identificar elementos que puedan ser reutilizados.

1.5.1 Sistema de Plug-ins de KDE

Como se mencionó con anterioridad, KDE es un entorno de escritorio “pesado”. Este brinda a los desarrolladores un marco de trabajo (*framework*) para la gestión de componentes que permite la creación de “partes[24]“ y “agregados[25]“ (SWEET, 2001). Dicho marco de trabajo constituye una extensión del mecanismo provisto por Qt mediante la incorporación de funcionalidades para la integración de componentes visuales (partes) en tiempo de ejecución. Tiene como desventaja que depende del resto del marco de trabajo de KDE y de Qt.

---

21 **Serializar:** transformar datos a un formato binario para que sean entendido por el componente al que se quiere transmitir.
22 **Deserialización:** transformar datos binarios a código de programación.
23 **Bibliotecas dinámicas:** archivos con código ejecutable que se cargan bajo demanda de un programa.
24 En el contexto de KDE se define como un componente con interfaz gráfica.
25 En el contexto de KDE se define como un componente sin interfaz gráfica.

1.5.2 Sistema de Plug-ins de Leechcraft

El proyecto Leechcraft tiene como objetivo la creación de un entorno de escritorio modular. Para ello fueron desarrolladas un conjunto de librerías similares a las de KDE. Comprende la gestión de dependencias entre módulos, pero carece de la mayoría de los mecanismos necesarios para el desarrollo de aplicaciones modulares.

1.5.3 Marco de trabajo OSGi

El marco de trabajo OSGi (*Open Services Gateway Initiative,* Iniciativa de Puerta de Enlace de Servicios Abiertos) es desarrollado por la homónima organización desde 1999. Este provee un framework para Java que permite el despliegue de aplicaciones extensibles y accesibles vía web conocidos como "*bundles*" (OSGI ALLIANCE, 2011).

Las funcionalidades del framework están divididas en las siguientes capas:

- Capa de seguridad
- Capa de módulos
- Capa de ciclo de vida
- Capa de servicio

La **capa de seguridad** garantiza un formato de empaquetado seguro y la interacción de manera controlada con el resto de la aplicación. Está diseñada para entornos controlados y es opcional.

La **capa de módulos** extiende las capacidades de la plataforma Java para el empaquetado, despliegue y validación de aplicaciones y componentes. Esta define una unidad de modularización denominada "*bundle*". Un "*bundle*" es un comprimido de clases Java y otros recursos que juntos brindan determinada funcionalidad. Estos pueden intercambiar funcionalidades a través de interfaces bien definidas.

La **capa de ciclo de vida** provee una API para controlar la seguridad y las operaciones del ciclo de vida de los "*bundles*". Esta capa se basa en las de seguridad y de módulos. Comprende la instalación, inicio, parada, actualización, desinstalación y monitoreo de los "*bundles*".

La **capa de servicio** define un modelo dinámico y colaborativo que se encuentra altamente integrado con la capa de ciclo de vida. El modelo servicio responde a las tareas de publicar, buscar y enlazar. Un servicio

es un simple objeto Java que es registrado bajo una o más interfaces con el registro de servicio. Los "*bundles*" pueden registrar servicios, buscarlos o recibir notificaciones cuando el estado del registro cambia.

Como se puede apreciar OSGi es una plataforma madura que cuenta con amplia aceptación en la industria informática.

1.5.4 Sistema de Plug-ins de Eclipse

El entorno de desarrollo integrado (IDE[26], en inglés *integrated development environment*) Eclipse es un sistema de software que posee un enfoque modular. Este es desarrollado utilizando el lenguaje de programación Java y a partir de su versión 3 sigue un enfoque similar a OSGi para la gestión de módulos. A diferencia de este, Eclipse llama a sus módulos "plug-ins[27]", aunque mantiene un concepto similar. Eclipse gestiona las dependencias a nivel de "plug-ins" y no a nivel de paquetes. Aunque la mayor diferencia entre ambos es que Eclipse no es capaz de instalar, actualizar o desinstalar "plug-ins" dinámicamente[28]. Es distribuido bajo licencia Apache versión 2 (RUBEL, 2006).

1.5.5 CTK Plugin Framework

El proyecto CTK (Common Tool Kit) tiene como objetivo facilitar la informatización de sistemas biomédicos. Es un conjunto de herramientas desarrolladas en C++ utilizando el marco de trabajo Qt. Uno de sus componentes principales es el marco de trabajo para la gestión de "plug-ins". La última versión del mismo cumple con la versión 4.3 del estándar OSGi. Este es desarrollado activamente bajo licencia Apache versión 2, por lo que puede ser utilizado libremente y sin costo alguno en otros proyectos (German Cancer Research Center, 2014).

1.5.6 Biblioteca CppMicroServices

La utilización de una implementación completa del marco de trabajo OSGi implica un considerable consumo de recursos, por su complejidad. Por ello surge la librería CppMicroServices con el objetivo de ganar en ligereza. Esta es desarrollada utilizando el lenguaje C++, proporciona una implementación completa de la capa de servicios OSGi y solo lo suficiente de la capa de módulo, permitiendo la carga y descarga de los mismos. Se centra en gestionar el ciclo de vida de los módulos y la dependencia entre ellos y no posee dependencias externas (Zelzer, 2013).

---

[26] **IDE:** es un programa informático compuesto por un conjunto de herramientas de programación.

[27] **Plug-in:** es una aplicación que se relaciona con otra para aportarle una función nueva. Esta aplicación adicional es ejecutada por la aplicación principal e interactúan por medio de la API.

[28] Mientras se ejecuta la aplicación.

1.5.7 Comparación de soluciones

Luego de analizadas las características de estos sistemas, se formula la siguiente tabla para el análisis y posterior selección de la solución más afín al desarrollado de este trabajo.

Sistemas	**Lenguaje Programación**	**Consumo de recursos**	**Gestión de componentes**	**Gestión de servicios**	**Dependencias externas**
OSGi Framework	Java	Alto	Si	Si	Ninguna, JRE[29]
Sistema de Plug-ins de Eclipse	Java	Alto	Si	Si	Ninguna, JRE
CTK Plugin Framework	C++	Bajo	Si	Si	Qt Framework 4
Biblioteca CppMicroServices	C++	Bajo	Si	Si	Ninguna
Sistema de Plug-ins de KDE	C++	Bajo	Si	No	KDE Framework Qt Framework 4
Sistema de Plug-ins de Leechcraft	C++	Bajo	Si	No	Qt Framework 4

Tabla 2: Comparación de soluciones

Según los datos ofrecidos en la tabla comparativa, este trabajo decidió utilizar la biblioteca CppMicroServices como base para el desarrollo de la solución propuesta, puesto que está orientada a proyectos desarrollados en el lenguaje C++, tiene un bajo consumo de recursos, permite la gestión de componentes, de servicios y no posee dependencias externas.

1.6 Metodología a utilizar

La metodología de desarrollo de software juega un papel importante en los procesos que involucran al software, ya que su aplicación garantiza el desarrollo de productos funcionales y libres de errores.

En este trabajo se hará uso de la **metodología OpenUp**, esta se caracteriza por ser ágil, se basa en casos de uso, en el desarrollo iterativo e incremental y es centrada en la arquitectura. Estructura el ciclo de vida de un proyecto en cuatro fases: Concepción, Elaboración, Construcción y Transición (OPENUP, 2012). A continuación una breve descripción de sus fases.

[29] **JRE:** *Java Runtime Environment,* Ambiente de ejecución de Java: incluye la máquina virtual de Java y las bibliotecas de Java.

- **Concepción:** se enfoca en la definición del propósito y los objetivos mediante la obtención de suficiente información sobre lo que se debe hacer en el proyecto. El objetivo de ésta fase es capturar las necesidades de los stakeholder[30] en los objetivos del proyecto.
- **Elaboración:** Consiste en tratar los riesgos significativos para la arquitectura. El propósito de esta fase es establecer la línea base para la elaboración de la arquitectura del sistema.
- **Construcción:** está enfocada al diseño, implementación y prueba de las funcionalidades del sistema. El propósito de esta fase es completar el desarrollo del sistema basado en la arquitectura definida.
- **Transición:** su propósito es asegurar que el sistema es entregado a los usuarios, y evaluar las funcionalidades y rendimiento.

A continuación las características por las que fue escogida esta metodología como guía del proceso de desarrollo de la solución propuesta:

- Es recomendada por el centro al que pertenece este trabajo.
- Está publicada bajo una licencia libre.
- Es un proceso mínimo y suficiente.
- Está diseñada para proyectos pequeños.

La metodología OpenUp en la realización del modelado del sistema, hace uso de las 4+1 vistas de Kruchten (KRUCHTEN, 1995).

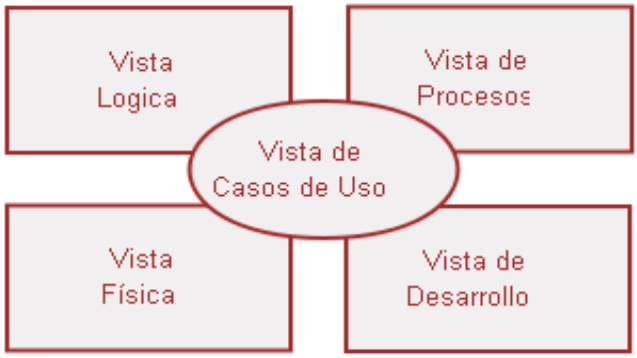

Ilustración 1: 4+1 Vistas de Kruchten

[30] **Stakeholder:** Involucrados: usuarios, clientes, desarrolladores, analista, arquitecto, etc.

Según lo expuesto por el ingeniero Philippe Kruchten, las distintas vistas del enfoque responden a las necesidades de las partes interesadas. Para cada uno de estas, se presenta una visión resumida del sistema con la información que requieren para satisfacer sus necesidades. A continuación se expone una breve descripción de estas (KRUCHTEN, 1995):

- **Vista de casos de uso:** Proyecta el comportamiento del sistema tal y como es percibido por los usuarios finales, analistas y encargados de las pruebas.

- **Vista lógica:** Soporta los requisitos funcionales del sistema: servicios proporcionados a los usuarios finales. Vocabulario del problema y su solución: clases, interfaces y colaboraciones.

- **Vista de procesos:** Cubre el funcionamiento, capacidad de crecimiento y rendimiento del sistema. Mecanismos de sincronización y concurrencia del sistema: hilos y procesos.

- **Vista de desarrollo:** Cubre la gestión de configuraciones de las distintas versiones de un sistema a partir de componentes y archivos quasi-independientes. Ensamblado y disponibilidad del sistema: componentes y archivos.

- **Vista de despliegue:** Contiene los nodos que forman la arquitectura (topología), hardware sobre el que se ejecuta el sistema a través de sus componentes. Está destinada a representar la distribución, entrega e instalación de las partes que forman el sistema informático físico.

Según lo planteado por el Ingeniero Kruchten, las vistas que no son útiles pueden omitirse de la descripción de la arquitectura, tales como la vista física si hay un único procesador, y la vista de procesos si existe un solo proceso o programa (KRUCHTEN, 1995). En este trabajo no se generan estas vistas, puesto que la solución propuesta está compuesta por un único proceso y es desplegada en un único computador (utiliza un procesador).

1.7 Herramientas a utilizar

A continuación se muestran las herramientas a utilizar durante el desarrollo de la solución propuesta por esta investigación:

Marco de trabajo QT 5.2: se utiliza para el desarrollo de software de aplicación con interfaz gráfica de usuario (GUI), y para el desarrollo de los programas no gráficos como herramientas de línea de comandos y consolas para servidores (Thelin, 2007).

Entorno de Desarrollo Integrado[31] (IDE) QTCreator 5.2: es un IDE creado por Trolltech para el desarrollo de aplicaciones con las bibliotecas Qt. Los sistemas operativos que soporta en forma oficial son: GNU/Linux 2.6.x, para versiones de 32 y 64 bits con Qt 5.x instalado. Este IDE tiene la ventaja de ser soportado por la misma compañía que soporta a Qt, por lo que propicia una mayor integración y aumenta el número de funcionalidades (Thelin, 2007).

Herramienta de modelado Visual Paradigm 10.1: Aunque es un software propietario; la universidad posee una licencia para el trabajo con dicha herramienta, puesto que es una de las más potentes que existen en la actualidad, multiplataforma y orientada a BPM (Gestión de Procesos de Negocio). Posee entre sus principales características la posibilidad de crear un amplio conjunto de artefactos, utilizados con frecuencia durante el desarrollo de software (VISUAL PARADIGM, 2013).

Herramienta de construcción Cmake 2.8: es una familia de herramientas diseñada para construir, probar y empaquetar software. Se utiliza para controlar el proceso de compilación del software usando ficheros de configuración sencillos e independientes de la plataforma. Cmake genera makefiles[32] nativos y espacios de trabajo que pueden usarse en el entorno de desarrollo deseado (CABRERA, 2009).

Sistema de control de versiones Git 1.9.1: sistema que registra los cambios realizados sobre un archivo o conjunto de archivos a lo largo del tiempo, de modo que se pueda recuperar versiones específicas en un futuro**.** Git es diseñado por Linus Torvalds, pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando estas tienen un gran número de archivos de código fuente (Git Project, 2013).

Herramienta de medición de tiempo de ejecución time 1.7: Mide el tiempo de ejecución de un programa. Toma la medida del tiempo durante el que se ha estado ejecutando la aplicación (real), y de los tiempos de ejecución de código en modo usuario (user) y en modo supervisor (sys), como resultados de llamadas del programa al sistema operativo (MACKENZIE, 2010).

Herramienta de análisis de software Valgrind 3.10: Marco de trabajo para el desarrollo de herramientas de análisis de software. Incluye módulos para la detección de pérdida de memoria y análisis de la pila (VALGRIND DEVELOPERS, 2013).

---

[31] **IDE:** es un entorno de programación creado como un programa de aplicación con un conjunto de herramientas para el programador.

[32] **Makefiles:** son ficheros de texto que utiliza make (herramienta de gestión de dependencias, entre los archivos que componen el código fuente de un programa) para llevar la gestión de la compilación de programas.

1.8 Lenguajes a utilizar

Lenguaje de programación C++ 98 : Lenguaje de programación libre y compilado. Es considerado como un lenguaje de nivel intermedio, ya que se compone de características de alto y bajo nivel. C++ es a la vez un lenguaje procedural (orientado a algoritmos) y orientado a objetos. Como lenguaje procedural se asemeja al C y es compatible con él. Como lenguaje orientado a objetos se basa en una filosofía completamente diferente, que exige del programador un completo cambio de mentalidad. Las características propias de la Programación Orientada a Objetos de C++ son modificaciones mayores que sí cambian radicalmente su naturaleza (SARRIEGUI, 1998).

Lenguaje de modelado UML 1.4: Es el lenguaje que permite la modelación de sistemas de software con tecnología orientada a objetos más conocido y utilizado en la actualidad. Es un lenguaje gráfico que cuenta con un grupo de diagramas, los cuales son utilizados para visualizar, especificar, construir y documentar un sistema de software en cada una de las etapas por las que tiene que pasar (RUMBAUCH, 2000).

1.7 Conclusiones del capítulo

El desarrollo de este capítulo permitió identificar las características sobre las cuales se desarrolla el Núcleo del EE de Nova Ligero 2015, las herramientas, lenguajes y metodología a utilizar, proponiéndose como solución:

- La implementación de una Arquitectura Orientada a Servicios, para permitir la interoperabilidad entre los componentes del EE de Nova Ligero 2015, mediante la publicación de sus datos y funcionalidades como servicios.
- Transformar las aplicaciones del EE de Nova Ligero 2015 en bibliotecas dinámicas (plug-in) que sean cargados por la misma aplicación, permite la comunicación de manera directa sin incurrir en la duplicación de datos.
- Hacer uso de la biblioteca CppMicroServices en la implementación de las funcionalidades de la Arquitectura Orientada a Servicios.

Capítulo 2: “DISEÑO E IMPLEMENTACIÓN DE LA SOLUCIÓN PROPUESTA”

La etapa de diseño de un sistema juega un papel importante durante el desarrollo de un software, gracias a las metodologías existentes y a la diversidad de diagramas es posible que la estructura de un software sea entendida por todos los involucrados en el proceso de desarrollo. En el presente capítulo se expone la solución propuesta y el modelado del sistema, haciendo uso de tres vistas de las propuestas en el modelo de las “4+1” vistas de Kruchten, utilizado por la Metodología OpenUp. Se exponen además el estilo arquitectónico, los principios de diseño, patrones de diseño y estándares de codificación utilizados durante el desarrollo del sistema.

2.1 Propuesta de solución

La propuesta de solución consiste en el desarrollo de un Núcleo para el EE de Nova Ligero 2015. El mismo es una aplicación que implementa una Arquitectura Orientada a Servicios e integra los componentes del EE bajo el concepto de biblioteca dinámica (plug-in). Mediante los perfiles de configuración se definen los componentes a ser cargados por el Núcleo. Luego, estos podrán interoperar mediante la publicación y consumo de datos y funcionalidades como servicios. Para una mayor comprensión de la solución, se expone a continuación el modelo de dominio del sistema.

2.2 Modelo de Dominio

El Modelo de Dominio (o Modelo Conceptual) es una representación visual de los principales conceptos u objetos del mundo real, significativos para un problema o área de interés. Este es de gran ayuda para desarrolladores y usuarios, de esta forma se utiliza un vocabulario común y pueden entender el contexto en que se enmarca el sistema (MARTINEZ, 2012).

2.2.1 Diagrama de clases del Modelo de Dominio

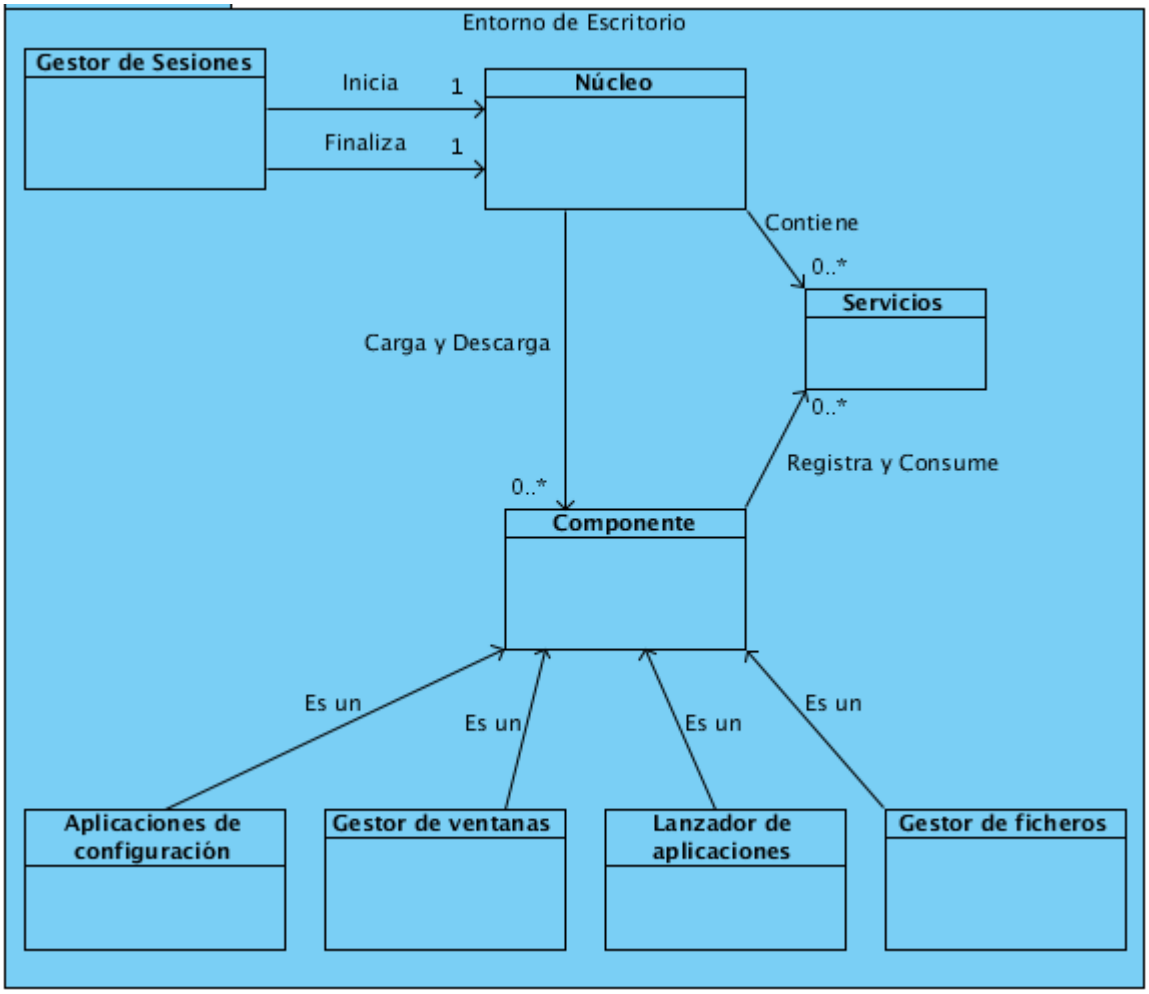

Ilustración 2: Diagrama de clases del Modelo de Dominio

2.2.2 Descripción de las clases del Modelo de Dominio

- **Gestor de sesión:** Concepto encargado de permitir el acceso al entorno de escritorio.
- **Núcleo:** Concepto encargado de proporcionar las funcionalidades necesarias para establecer la interoperabilidad entre componentes del entorno de escritorio.
- **Servicios:** Concepto compuesto por datos y funcionalidades.
- **Componente:** Concepto que ofrece al usuario funcionalidades que pueden ser accedidas mediante una interfaz gráfica.

- **Aplicaciones de configuración:** Concepto que ofrece funcionalidades para realizar configuraciones al sistema.
- **Gestor de ventana:** Concepto encargado de ofrecer funcionalidades para cambiar el estado de las ventanas.
- **Lanzador de aplicaciones:** Concepto que ofrece la funcionalidad para ejecutar aplicaciones.
- **Gestor de ficheros:** Concepto encargado de brindar funcionalidades para explorar carpetas y relacionar ficheros con programas para su reproducción y gestionar los eventos de arrastrar.

2.3 Modelado del Sistema

Con el objetivo de realizar el modelado del sistema de forma eficiente y que cumpla con las expectativas de los involucrados, se hace uso de la metodología OpenUp y se aplican la vista de casos de uso, vista lógica y vista de desarrollo. Por otra parte OpenUp define en su primera fase la identificación de los requisitos que debe cumplir el sistema a desarrollar. A continuación se muestran los requisitos identificados mediante las técnicas de captura de requisitos: Estudio de homólogos y Tormenta de ideas.

2.3.1 Requisitos del sistema

Un requisito es una condición o capacidad que debe tener un sistema para satisfacer las necesidades de un cliente, con el fin de resolver un problema planteado por el mismo en forma de un documento formal (IEEE, 2009). Estos se dividen en Requisitos Funcionales(RF) y Requisitos No Funcionales (RNF).

2.3.1.1 Requisitos funcionales

Los requisitos funcionales definen las funciones que es capaz de realizar el sistema, es decir, describen las funcionalidades o los servicios que se espera que este provea (RUMBAUCH, 2000). A continuación los requisitos funcionales del sistema:

RF_1 Iniciar utilizando un perfil de configuración.

RF_2 Cargar componentes.

RF_3 Descargar componentes.

RF_4 Crear componentes.

RF_5 Modificar componentes.

RF_6 Desplegar componentes.

RF_7 Notificar cuando se carguen los componentes de inicio.

RF_8 Notificar cambios en las variables de ejecución.

RF_9 Registrar servicios.

RF_10 Buscar servicios.

RF_11 Utilizar servicios.

RF_12 Gestionar variables de entorno de ejecución.

RF_13 Notificar el cierre del sistema.

2.3.1.2 Requisitos no funcionales

Los requisitos no funcionales son aquellos que no se refieren directamente a las funciones que debe realizar el sistema, sino a las características que puedan limitarlo de una forma u otra (RUMBAUCH, 2000). A continuación los requisitos no funcionales del sistema:

RNF_1 Consumir menos de 3 MiB[33] de RAM.

RNF_2 Cargar en menos de 500 ms[34].

2.3.2 Vista de Casos de Uso

Para una mayor comprensión de las funcionalidades que posee el sistema se exponen a continuación los actores y casos de uso, con sus respectivas descripciones. Artefactos[35] mediante los cuales se guió el diseño del sistema. En la sección Anexos (solo en la versión digital del documento) se podrá consultar el resto de las descripciones de los casos de uso.

[33] **Mebibyte:** unidad de almacenamiento de información equivalente a 2^{20} bytes. Definido en el estándar IEC 80000-13:2008.
[34] **Milisegundo:** Medida de tiempo que es igual a la milésima parte de un segundo.
[35] **Artefacto:** Cualquier cosa que resulte del proceso de desarrollo de software; por ejemplo: documentos de requisitos, especificaciones, diseños, software, etc.

2.3.2.1 Actores del Sistema

Actores	Descripción
Gestor de sesión	Aplicación encargada de representar la sesión.
Componente	Aplicaciones del entorno de escritorio.
Desarrollador de componentes	Persona encargada de desarrollar las aplicaciones del entorno de escritorio.
Integrador del escritorio	Persona encargada de definir los componentes que deben ser cargados por el Núcleo.

Tabla 3: Descripción de los actores del sistema

2.3.2.2 Diagrama de Casos de Uso del Sistema

Los casos de uso son artefactos narrativos que describen, bajo la forma de acciones y reacciones, el comportamiento del sistema desde el punto de vista del usuario. Establece un acuerdo entre clientes y desarrolladores sobre las condiciones y posibilidades (requisitos) que debe cumplir el sistema (CERIA, 2011).

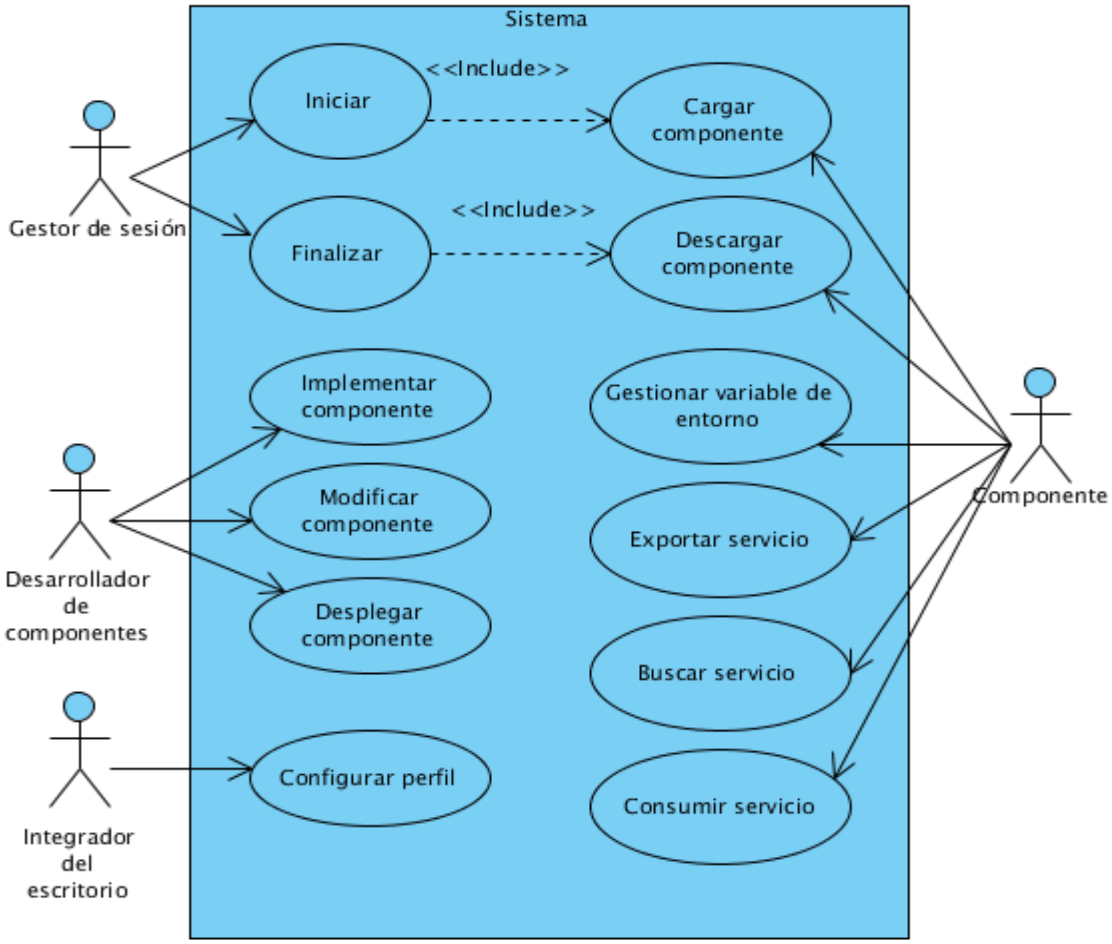

Ilustración 3: Casos de uso del sistema

2.3.2.3 Descripción de los Casos de Uso

Objetivo	Cargar componentes	
Actores	Componente, Gestor de sesión	
Resumen	El sistema carga la biblioteca compartida que contiene el componente especificado.	
Complejidad	Alta	
Prioridad	Crítico	
Precondiciones		
Postcondiciones	El componente es cargado y activado.	
Flujo de eventos		
Flujo básico “Cargar Componente”		
	Actor	**Sistema**
1	Solicita que se cargue un componente indicando su nombre.	
2		Verifica que el componente no esté cargado.
3		Busca la biblioteca compartida que corresponde al componente en las rutas configuradas.
4		Carga la biblioteca compartida.
5		Ejecuta el activador del componente.
6		Termina el caso de uso.
Flujos alternos		
1.“El componente ya está cargado”		
	Actor	**Sistema**
1	Solicita que se cargue un componente indicando su nombre.	
2		Verifica que el componente no esté cargado.
3		Termina el caso de uso.
Flujos alternos		
1.“La biblioteca compartida no existe o no está disponible”		
	Actor	**Sistema**
1	Solicita que se cargue un componente	

	indicando su nombre.	
2		Verifica que el componente no esté cargado.
3		Busca la biblioteca compartida que corresponde al componente en las rutas configuradas.
4		Emite una notificación sobre el suceso.
5		Termina el caso de uso.

Tabla 4: Descripción del caso de uso Cargar componentes

Objetivo	Descargar componentes	
Actores	Componente, Gestor de sesión	
Resumen	Se desactiva y descarga de memoria el componente.	
Complejidad	Alta	
Prioridad	Crítico	
Precondiciones		
Postcondiciones	El componente es descargado y desactivado.	
Flujo de eventos		
Flujo básico “Solicitar servicio”		
	Actor	**Sistema**
1	Solicita que el componente sea descargado.	
2		Revisa que los servicios registrados por el componente no estén en uso.
3		Se ejecuta el desactivador del componente.
4		Se libera la memoria correspondiente a la biblioteca compartida.
5		Termina el caso de uso.
Flujos alternos		
1. “Los servicios del componente están en uso”		
	Actor	**Sistema**
1	Solicita que el componente sea descargado.	
2		Revisa que los servicios registrados por el componente no estén en uso.
3		Emite una notificación sobre el suceso.
4		Termina el caso de uso.

Tabla 5: Descripción del caso de uso Descargar componentes

Objetivo	Buscar servicio	
Actores	Componente.	
Resumen	El componente busca los servicios disponibles para determinada interfaz.	
Complejidad	Media	
Prioridad	Crítico	
Precondiciones	El componente conoce la interfaz del servicio.	
Postcondiciones	Se obtiene un conjunto de referencias a los servicios disponibles.	
Flujo de eventos		
Flujo básico “Buscar servicio”		
	Actor	**Sistema**
1	Define el patrón de búsqueda a utilizar.	
2	Solicita los descriptores del servicio especificando la interfaz del servicio y el patrón de búsqueda.	
3		Obtiene una lista de referencias a los servicios relacionados con la interfaz especificada.
4		Filtra la lista de referencias utilizando el patrón de búsqueda.
5		Devuelve la lista de referencias a servicios filtrada.
6		Termina el caso de uso.
Flujos alternos		
1. “No existen servicios que cumplas con el patrón de búsqueda”		
	Actor	**Sistema**
1	Define el patrón de búsqueda a utilizar.	
2	Solicita los descriptores del servicio especificando la interfaz del servicio y el patrón de búsqueda.	
3		Obtiene una lista de referencias a los servicios relacionados con la interfaz especificada.
4		Filtra la lista de referencias utilizando el patrón de

		búsqueda.
5		Devuelve la lista de referencias a servicios vacía.
6		Termina el caso de uso.

Tabla 6: Descripción del caso de uso Buscar servicios

Objetivo	Consumir servicio	
Actores	Componente	
Resumen	El componente consume un servicio.	
Complejidad	Alta	
Prioridad	Crítico	
Precondiciones	El componente posee una referencia al servicio.	
Postcondiciones		
Flujo de eventos		
Flujo básico “Solicitar servicio”		
	Actor	**Sistema**
1	Solicita el objeto del servicio.	
		Se incrementa el contador de referencia al servicio.
2		Retorna un puntero al objeto.
3	Utiliza los métodos definidos en la interfaz del servicio.	
4	Libera la referencia al servicio	
5		Se decrementa el contador de referencias al servicio.
6		Si el contador llega a 0 se destruye elimina el servicio.
7		Termina el caso de uso.

Tabla 7: Descripción del caso de uso Consumir servicio

Objetivo	Iniciar sistema	
Actores	Gestor de sesión	
Resumen	El gestor de sesión inicia la aplicación, especificando el perfil y la ruta donde se encuentran instalados los componentes a utilizar, mediante argumentos.	
Complejidad	Alta	
Prioridad	Crítico	
Precondiciones	Se ha iniciado una sesión.	
Postcondiciones	Se cargaron los componentes especificados en el perfil de configuración.	
Flujo de eventos		
Flujo básico “Iniciar sistema”		
	Actor	**Sistema**
1	Ejecuta la aplicación definiendo el perfil de configuración a utilizar.	
2		Obtiene el perfil de configuración.
3		Extrae la lista de componentes definidos en el perfil.
4		Carga secuencialmente los componentes. Ver caso de uso: “Cargar Componente”
5		Emite una señal para indicar que se terminó la carga de los componentes.
6		Termina el caso de uso.
Flujos alternos		
1.“No se especifica el perfil de configuración”		
	Actor	**Sistema**
1	Ejecuta la aplicación	
2		Define como perfil de configuración “Default”.
3		Continua el “**Flujo básico**”, a partir del paso 2.
2.“No existe el perfil de configuración”		
	Actor	**Sistema**
1	Ejecuta la aplicación	

2		El sistema termina con valor 10.	
3		Termina el caso de uso.	
Relaciones	**CU Incluidos**	Cargar componentes	

Tabla 8: Descripción del caso de uso Iniciar

Objetivo	Finalizar el sistema		
Actores	Gestor de sesión: finaliza el sistema		
Resumen	Cuando el sistema es finalizado por el Gestor de sesión se descargan los componentes que conformaban el EE.		
Complejidad	Baja		
Prioridad	Crítico		
Precondiciones	El sistema ha sido iniciado		
Postcondiciones	Se descargaron los componentes del EE.		
Flujo de eventos			
Flujo básico “Finalizar sistema”			
	Actor	**Sistema**	
1	Finaliza la aplicación		
2		Descarga los componentes secuencialmente, consultando la lista de componentes cargados. Ver caso de uso: “Descargar componentes”	
		Emite una señal para indicar que se terminó la descarga de los componentes.	
		Termina el caso de uso.	
Relaciones	**CU Incluidos**	Descargar componentes	

Tabla 9: Descripción del caso de uso Finalizar

Objetivo	Exportar servicio
Actores	Componente
Resumen	El componente exporta un servicio a través de una interfaz definida
Complejidad	Alta
Prioridad	Crítico
Precondiciones	

Postcondiciones	El servicio está disponible en el sistema	
Flujo de eventos		
Flujo básico “Exportar servicio”		
	Actor	**Sistema**
1	Crea el objeto que será exportado como servicio.	
2	Crea el descriptor del servicio.	
3	Solicita el registro del objeto especificando la interfaz y las propiedades.	
4		Agrega la referencia del objeto al registro.
5		Termina el caso de uso.

Tabla 10: Descripción del caso de uso Exportar servicio

2.4 Propuesta de la Arquitectura del sistema

La ausencia de interoperabilidad entre componentes constituye la principal deficiencia de la arquitectura actual del EE de Nova ligero. En el capítulo 1 fueron analizados los mecanismos a utilizar para lograr este fin, identificándose como mejor alternativa el uso del estilo arquitectónico Orientado a Servicios. Este aparte de permitir el intercambio de datos y funcionalidades entre componentes facilita la integración de software desarrollado por terceros y brinda flexibilidad al sistema mediante la creación de capas de abstracción entre los clientes y los proveedores de servicios.

Para la aplicación de dicho estilo arquitectónico se siguió el estándar definido por la *OSGi Alliance* (Alianza OSGi), específicamente se utilizaron las secciones referentes al manejo de componentes y de servicios. Según el estudio de homólogos realizado la biblioteca *CppMicroServices* constituye, entre las diferentes implementaciones del mencionado estándar, la mejor alternativa a utilizar en el EE de Nova Ligero por no tener dependencias externas, estar implementada utilizando el lenguaje C++ y solo proveer las funcionalidades de gestión de componentes y servicios.

De acuerdo con la implementación del estándar OSGi para C++ un componente es una biblioteca compartida que posee un identificador único e implementa los métodos *Load* (Cargar) y *Unload* (Descargar). El método “*Load*” se ejecuta cuando el componente es cargado en memoria. Este debe encargarse de inicializar y publicar los servicios del componente. El método *“Unload*” se ejecuta antes de

ser descargado el componente. Este debe liberar la memoria reservada y eliminar del registro sus servicios.

Los servicios son definidos como un conjunto de tres elementos: una interfaz, un objeto que implemente la interfaz y una lista de propiedades que identifican al servicio. Estos datos son almacenados en el registro de servicios agrupándolos por la interfaz. El registro de servicios actúa como intermediario entre componentes y permite recuperar los servicios de una interfaz determinada filtrados por sus propiedades. De esta manera para comunicar un componente con otro solo es necesario conocer la interfaz del servicio a consumir (ver Ilustración 4).

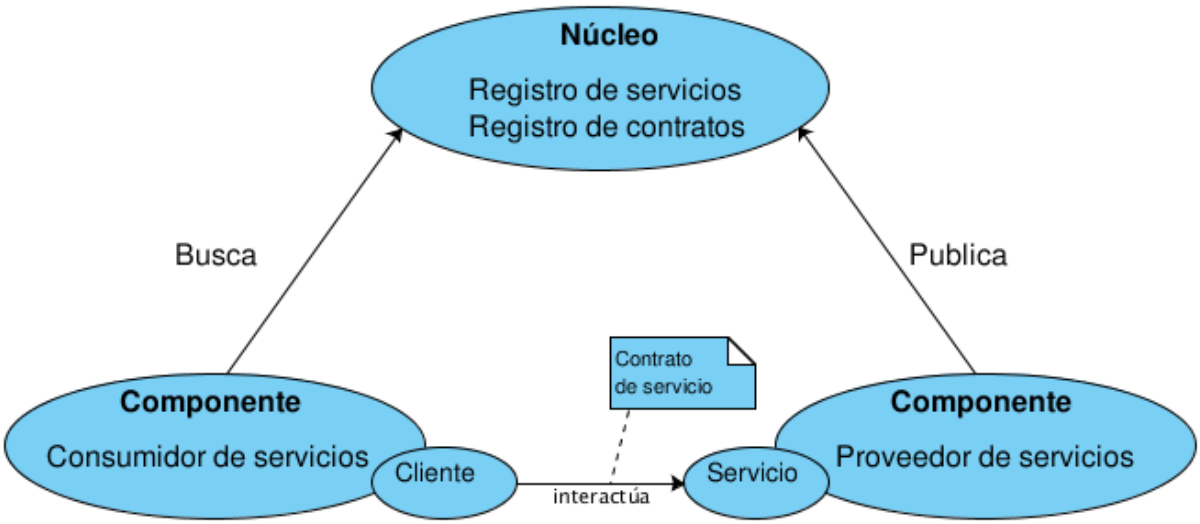

Ilustración 4: Interoperabilidad entre componentes del sistema

Una vez que un componente solicita un servicio se establece un contrato entre el proveedor y el consumidor. Este contrato se utiliza para controlar las relaciones entre componentes en tiempo de ejecución e impedir la descarga de un componente cuando sus servicios están siendo utilizados o en caso contrario descargarlo de manera automática para liberar memoria (ver Ilustración 5).

El código de una biblioteca compartida no puede ser ejecutado de manera independiente por ello fue preciso desarrollar una aplicación que utilice las funcionalidades de *CppMicroServices*. Esta aplicación fue nombrada Núcleo, la cual no se debe confundir con el *kernel* Linux[36]. La misma utiliza las funcionalidades de *CppMicroServices* para la gestión de servicios y extiende las funcionalidades de gestión de componentes añadiendo un mecanismo para su localización en el sistema. Además brinda servicios

[36] **Kernel Linux**: Sistema operativo de código abierto.

elementales para los componentes como son: gestión de la configuración y gestión de variables de entorno.

El Núcleo por si solo no provee funcionalidad alguna al usuario, esta tarea es desempeñada por los componentes. En dependencia de los servicios que proveen los componentes se agrupan en tres categorías: comunes, componentes gráficos y aplicaciones embebidas. Los componentes comunes son aquellos que proveen servicios que serán utilizados por varios componentes y no generan interfaces de usuario. Los componentes gráficos constituyen los ladrillos de construcción de las aplicaciones embebidas al proveer elementos gráficos para su construcción. Por último las aplicaciones embebidas[37] mediante la agrupación de varios componentes gráficos, brindan un conjunto de funcionalidades integradas, al usuario. El conjunto de todos los componentes antes mencionados y sus servicios constituyen la **capa de servicios** del EE (ver Ilustración 5).

La capa de servicios requiere la existencia del Núcleo, del servicio de D-Bus para la comunicación con aplicaciones de terceros y del gestor de sesión para activar o reiniciar los componentes anteriores en caso de fallos. Además utiliza varias bibliotecas desarrolladas por terceros para la interacción con el sistema y la creación de interfaces gráficas. Estos componentes dependen del sistema y del hardware por lo que limitan la portabilidad y reusabilidad del software. Para contrarrestar esta situación los componentes antes mencionados son separados en una capa independiente, capa de infraestructura, para que los cambios en la base del EE tengan un impacto limitado en los componentes desarrollados.

[37] **Aplicaciones embebidas:** Aplicaciones que comparten un mismo hilo de ejecución pero se muestran en ventanas.

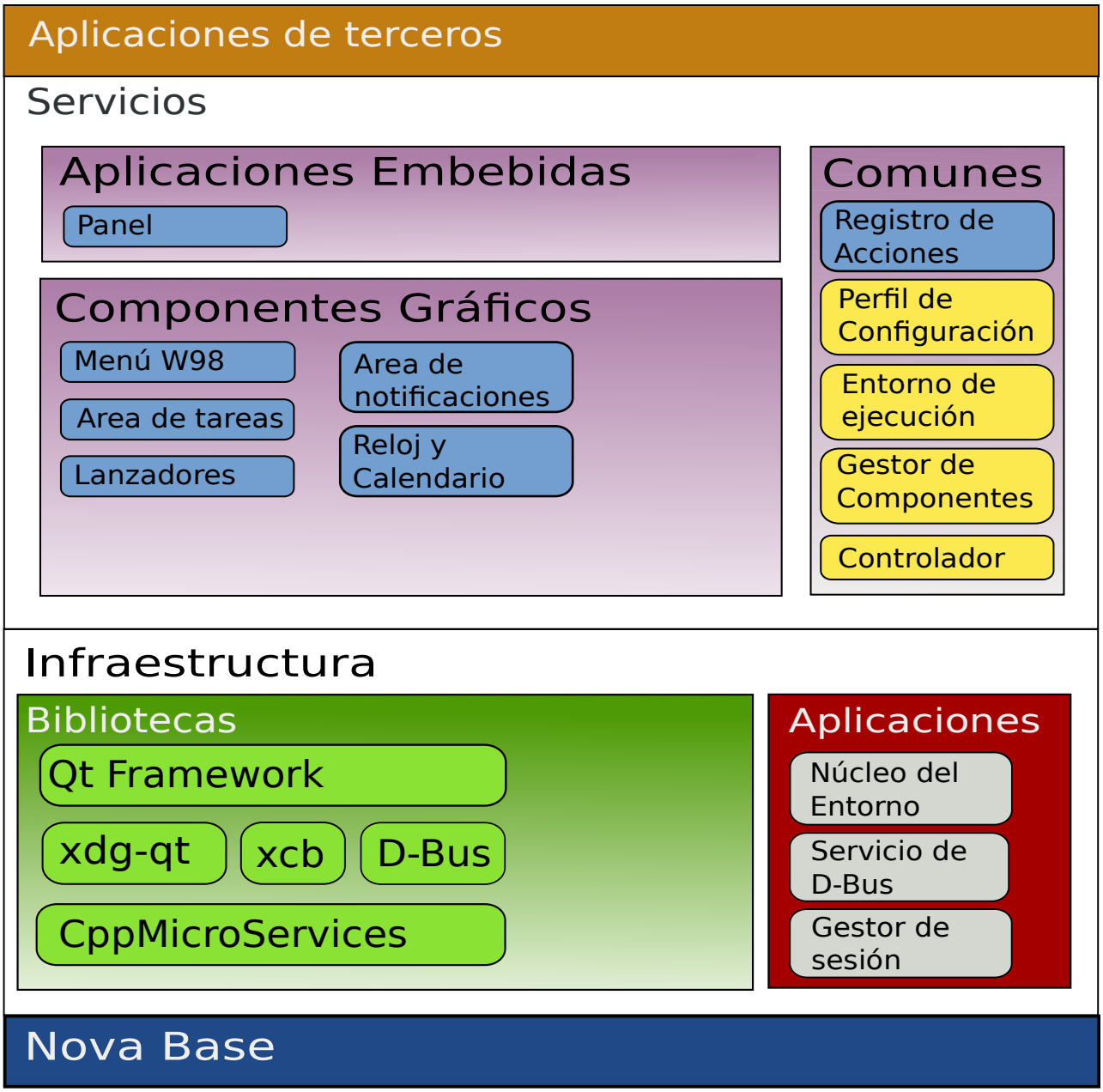

Ilustración 5: Propuesta parcial de la arquitectura para el EE de Nova Ligero 2015

De manera general se propone de acuerdo al estilo N-Capas separar el EE en dos capas: servicios e infraestructura. En la capa de servicios se implementará la lógica del sistema y en la capa de infraestructura se colocarán los componentes dependientes del sistema o desarrollados por terceros. Sobre el EE serán desplegadas las aplicaciones de terceros este a su vez se soporta en el sistema base. La comunicación dentro del EE se realizará mediante servicios de acuerdo al estándar OSGi y su implementación para C++ *CppMicroServices.* Para la comunicación con las aplicaciones de terceros y con el sistema se utilizará D-Bus.

2.5 Vista Lógica

2.5.1 Patrones de diseño

En el diseño del sistema se utilizan los principios de diseño **"SOLID"** (MCLAUGHLIN, 2007), sus siglas son desglosadas a continuación:

- **S**ingle Responsibility Principle (Principio de única responsabilidad): Consiste en que una clase debería concentrarse solo en hacer una cosa de tal forma que cuando cambie algún requisito en mayor o menor medida dicho cambio solo afecte a dicha clase por una razón.
- **O**pen / Closed Principle (Principio abierto/cerrado): Consiste en que las entidades de software (clases, módulos, funciones, etcétera) deberían estar abiertas a la extensión pero cerradas a la modificación. Es anticíparse al cambio, preparar el código para que los posibles cambios en el comportamiento de una clase se puedan implementar mediante herencia y composición.
- **L**iskov Substitution Principle (Principio de sustitución de Liskov): Consiste en que las subclasses deben comportarse adecuadamente cuando sean usadas en lugar de sus clases base.
- **I**nterface Segregation Principle (Principio de Segregación de Interfaces): Consiste en que los clientes no deberían ser forzados a depender de interfaces que no utilizan. Se deben establecer interfaces pequeñas y cohesivas, que puedan coexistir unas con otras.
- **D**ependency Inversion Principle (Principio de Inversión de Dependencias): Consiste en hacer que las dependencias se establezcan entre abstracciones y no de concreciones. Recomendándose el uso de interfaces y clases abstractas.

El uso de estos principios garantiza que:

- Las clases del sistema respondan a una sola responsabilidad y por tanto tengan solo una razón para cambiar.

- El comportamiento de las clases puedan ser extendido mediante herencia sin afectar el código de la clase en cuestión.
- No existan incongruencias funcionales entre clases hijas y padres.
- Las interfaces respondan a las necesidades específicas de los clientes.
- Se establezcan dependencias a elementos del sistema menos propensos a cambios (clases abstractas o interfaces).

Además de los principios SOLID, se aplicaron en el diseño de clases, los patrones generales de software para asignación de responsabilidades, también conocidos como patrones **"GRASP"** (MCLAUGHLIN, 2007):

- Alta cohesión: Consiste en lo coherente que es la información que almacena una clase con las responsabilidades y relaciones que ésta tiene con otras clases.
- Bajo acoplamiento: Indica lo vinculadas que están unas clases con otras, lo que afecta un cambio en una clase a las demás y lo dependientes que son unas clases de otras.
- Creador: Consiste en identificar quién debe ser el responsable de la creación o instanciación de nuevos objetos o clases.
- Experto en información: Plantea que la responsabilidad de la creación de un objeto o la implementación de un método, debe recaer sobre la clase que conoce toda la información necesaria para crearlo o ejecutarlo.
- Fabricación pura: Consiste en la creación de clases que no representan un ente u objeto real del dominio del problema, sino que se crean intencionadamente para disminuir el acoplamiento, aumentar la cohesión y/o potenciar la reutilización del código.

2.5.2 Descripción de las clases del sistema

La solución propuesta consta de cuatro clases fundamentales: *Controller*, *SettingsProfile*, *ModuleManager* y *Environment*. En correspondencia existe igual número de interfaces para permitir la comunicación con el resto de los componentes del sistema (Ver Ilustración 6). De manera general las clases del sistema siguen los principios: responsabilidad única, para garantizar la cohesión y principio abierto/cerrado para facilitar la evolución y mantenimiento del sistema.

La clase ***Controller*** (en español Controlador) gestiona el flujo de inicio y terminación de la aplicación. Por ello, de acuerdo con los patrones "experto en información" y "creador", comprende los métodos "start" y

“finish” en los que se crean y destruyen las instancias de las clases *SettingsProfile, ModuleManager* y *Environment.* Las instancias de las clases creadas son exportadas como servicios utilizando la interfaz ModuleContext de la biblioteca CppMicroServices.

La clase ***ModuleManager*** (en español Gestor de módulos) responde a la gestión de componentes, permite su localización, carga y descarga. Esta clase no tiene dependencias de acuerdo con el patrón “bajo acoplamiento” y encapsula todos los métodos referentes a la gestión de módulos cumpliendo con el patrón “alta cohesión”.

La clase ***Environment*** (en español Entorno) brinda un mecanismo de notificación de los cambios que se producen en las variables del entorno de ejecución. Constituye un mecanismo simple para la comunicación entre componentes. Esta clase requiere las funcionalidades de *SettingsProfile* pero, en lugar de depender directamente de la misma estable la relación mediante la interfaz ISettingsProfile de acuerdo al principio de diseño inversión de dependencias.

La clase ***SettingsProfile*** (en español Perfil de configuraciones), tiene una función utilitaria ya que actúa como intermediario entre los componentes y sus ficheros de configuración permitiendo que un componente tenga tantas configuraciones como perfiles existan.

Por último las interfaces ***IController***, ***IModuleManager***, ***ISettingsProfile*** e ***IEnvironment*** fueron diseñadas según el principio de segregación de interfaces. Estas actúan como mecanismo de abstracción entre la difinición de los servicios y su implementación, de acuerdo con el patrón “fabricación pura”. Adicionalmente se tuvo en cuenta el principio de sustitución de Liskov para garantizar la no existencia de incoherencias en su comportamiento.

2.5.3 Estándares de codificación

Los estándares de codificación (HOFF, 2008) son definidos como:

> **“** Pautas de programación que no están enfocadas a la lógica del programa, sino a su estructura y apariencia física para facilitar la lectura, comprensión y mantenimiento del código.**”**

Durante la implementación se hace uso de los estándares emitidos para el lenguaje C++, donde:

- Se comentan los métodos implementados, explicando su funcionamiento.
- Los nombres de los identificadores, son nombres significativos para que por su simple lectura, pueda conocerse su función.

- Se aplica la nomenclatura *UpperCamelCase* en los nombres de clases, la cual consiste en utilizar las mayúsculas como separadores de palabras, iniciando cada nombre de identificador con mayúscula.
- Se aplica la nomenclatura *lowerCamelCase* en los nombres de métodos y variables, la cual consiste en utilizar letra minúscula en la primera palabra.

2.5.4 Diagrama de Clases

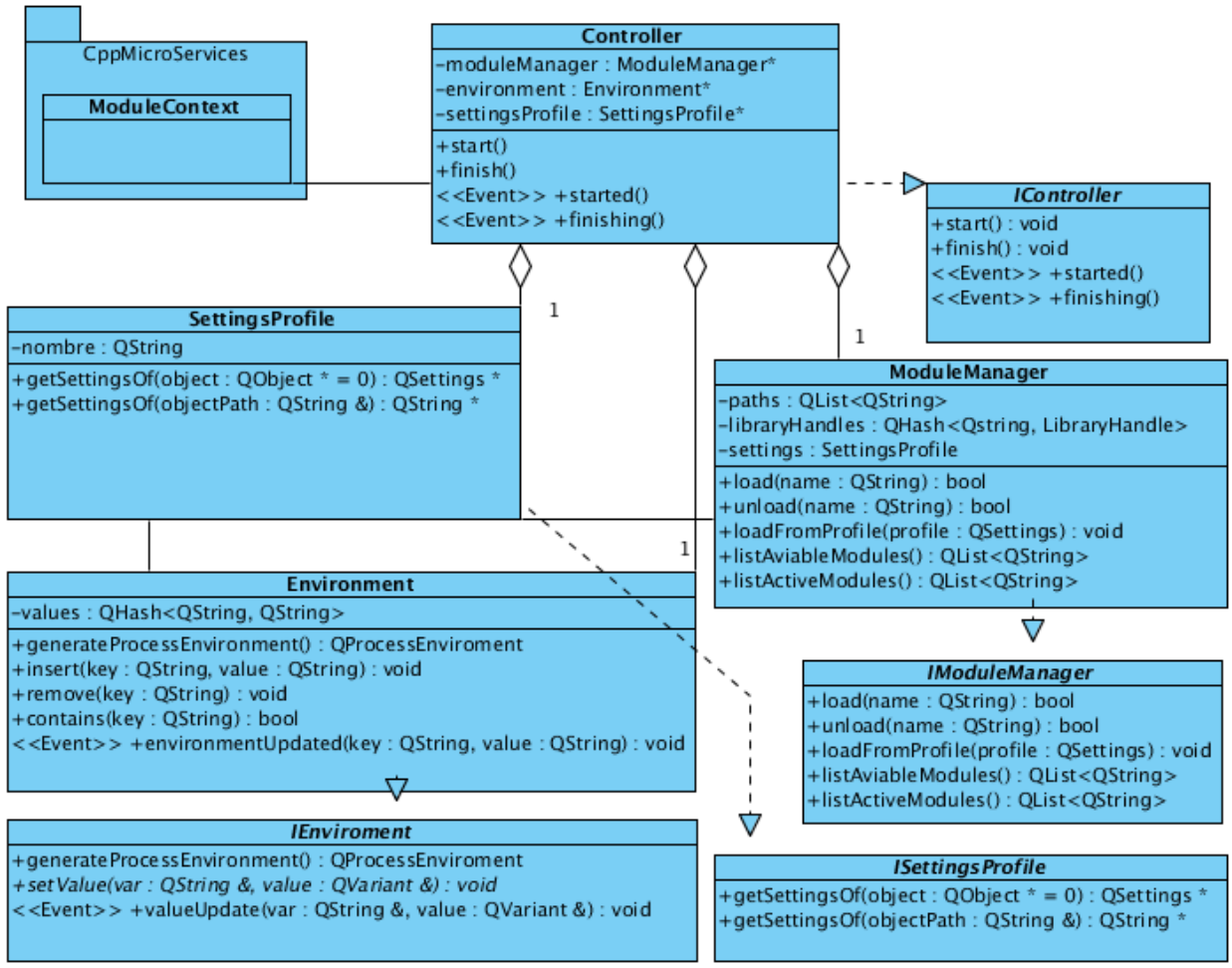

Ilustración 6: Diagrama de clases

2.6 Vista de Desarrollo

2.6.1 Diagrama de componentes

El diagrama de componentes está compuesto por el Gestor de sesiones (Session Manager), el Núcleo (Core) y la biblioteca CppMicroServices. El Núcleo implementa las interfaces *IController, IModuleManager, ISettingsProfile* e *IEnvironment* y utiliza la interfaz *ModuleContext* para acceder a las funcionalidades ofrecidas por la biblioteca CppMicroServices. El Gestor de sesiones utiliza la interfaz *IController* para el inicio y fin del sistema. El resto de las interfaces (*IModuleManager, ISettingsProfile, IEnvironment*) serán utilizadas por los componentes del EE, para acceder a los datos y funcionalidades que ofrece el Núcleo. (Ver Ilustración 7)

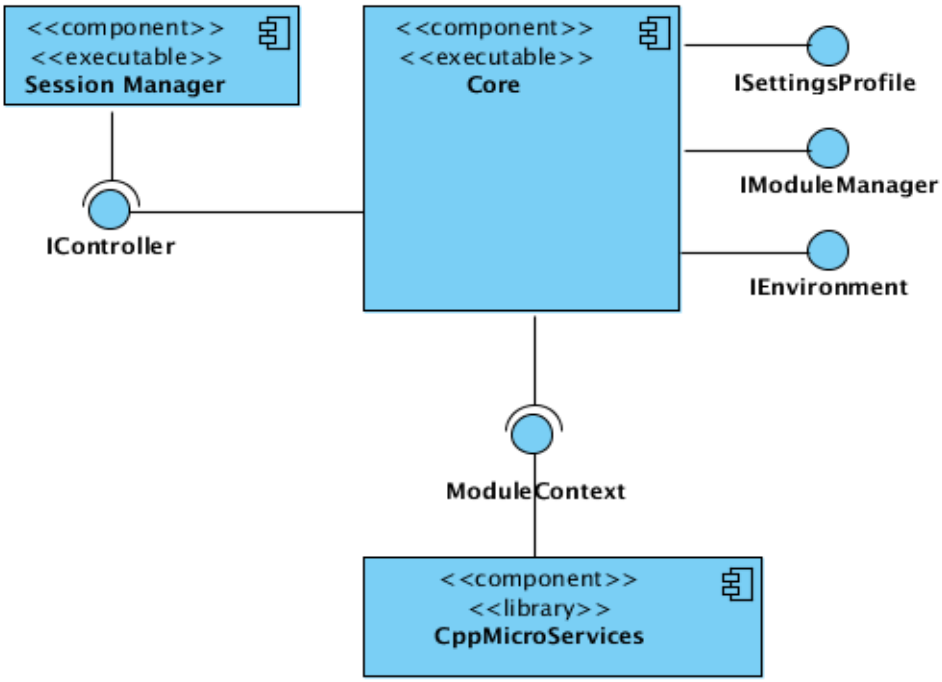

Ilustración 7: Diagrama de componentes

2.7 Conclusiones del capítulo

Con el desarrollo de este capítulo se realizó el modelado e implementación del sistema teniendo como guía la metodología OpenUp. Donde se obtuvo una versión funcional del sistema, lista para ser sometida al proceso de validación. Se generaron los artefactos necesarios para la implementación de la solución, lográndose el desarrollo de las funcionalidades que dan cumplimiento a los requisitos del sistema.

Capítulo 3: “VALIDACIÓN DE LA SOLUCIÓN PROPUESTA”

Para garantizar el correcto funcionamiento del sistema desarrollado, es imprescindible la realización de pruebas. La metodología OpenUp recomienda (EPF Project, 2014) en este sentido probar los requisitos funcionales críticos del sistema. Estos son probados mediante pruebas unitarias de caja blanca del tipo automatizadas y caja negra utilizando la interfaz de líneas de comandos. A continuación se muestran los requisitos críticos:

- Iniciar utilizando un perfil de configuración.
- Cargar componentes.
- Descargar componentes.
- Crear componentes.
- Modificar componentes.
- Desplegar componentes.
- Registrar servicios.
- Buscar servicios.
- Utilizar servicios.
- Gestionar variables de entorno de ejecución.

Las pruebas automatizadas son realizadas por herramientas de software desarrolladas a la medida para el producto. Para este fin el marco de trabajo Qt provee la biblioteca *QtTest* (QT Project, 2013), que agrupa un conjunto de funcionalidades para la creación, ejecución y mantenimiento de las herramientas de pruebas.

3.1 Perfil de configuración para las pruebas

La ejecución de las pruebas requiere configurar la aplicación para que se cargue al inicio el componente de pruebas. Por ello se utilizó como perfil de configuración el siguiente:

```
[Modules]
size=1
1\Name=moonlightDE-core_test
```

Ilustración 8: Perfil de configuración

Este se ubica en la ruta “~/.config/MoonLightDE/Test.conf”. Una vez configurado se ejecuta la aplicación utilizando el comando: “*moonlightDEcore -p Test*”.

3.2 Validación de las interfaces del Núcleo

La validación de las interfaces del Núcleo del EE se realizó mediante el desarrollo de un componente de prueba. Este prueba cada interfaz de manera independiente mediante la utilización de sus métodos y la verificación de los datos retornados. En el desarrollo del componente se hizo énfasis en la documentación de manera que sirva de referencia para los desarrolladores.

El componente de prueba se compone por dos clases: *Activator* y *TestICore*. La primera constituye el activador del componente y es instanciada cuando este se carga. Contiene dos procedimientos *Load* y *Unload.* El procedimiento *Load* gestiona la inicialización del componente instanciando la clase *TestICore* y ejecutando las pruebas. Por su parte *Unload* finaliza el componente liberando la memoria reservada.

En la clase *TestICore* se encuentra la lógica de las pruebas, estas se agrupan atendiendo a la interfaz que se valida. A continuación se describe el procedimiento utilizado en cada interfaz.

3.2.1 Interfaz *IController*

La interfaz *IController* es la principal interfaz del Núcleo y controla su inicio y finalización. Esta se relaciona con los requisitos funcionales: iniciar utilizando un perfil de configuración, notificar cuando se carguen los componentes de inicio, notificar el cierre del sistema, registrar servicios, buscar servicios y utilizar servicios. A continuación se muestra el código utilizado, en él se incluyen como comentarios los pasos que se realizan:

```
// Obtener y validar el contexto del componente.
ModuleContext * context = GetModuleContext();
QVERIFY(context != NULL);

// Obtener y validar la referencia al servicio publicado por el Núcleo.
ServiceReference<Core::IController>iControllerRef = context-
>GetServiceReference<Core::IController>();
Core::IController * iController = context->GetService(iControllerRef);
QVERIFY( iController != NULL);

// Conectar las señales del servicio y  validar su emisión.
QObject::connect(iController, SIGNAL(started()), this, SLOT(noticeLoadEnded()));
QObject::connect(qApp, SIGNAL(aboutToQuit()), this, SLOT(aboutToQuit()));
```

Ilustración 9: Prueba interfaz IController

3.2.2 Interfaz *IModuleManager*

Mediante la interfaz *IModuleManager* se realiza la gestión de los componentes en el sistema. La misma responde a los requisitos funcionales: cargar componentes, descargar componentes, registrar servicios, buscar servicios y utilizar servicios. Para validar su correcto funcionamiento fue necesario desarrollar un tercer componente, que será cargado y descargado. A continuación se expone el código de validación comentado:

```
// Obtener y validar el contexto del componente.
ModuleContext * context = GetModuleContext();
QVERIFY(context != NULL);

// Obtener el descriptor de servicio.
ServiceReference<Core::IModuleManager> iModuleManagerRef =
            context->GetServiceReference<Core::IModuleManager>();

// Obtener el servicio.
Core::IModuleManager * moduleManager = context->GetService(iModuleManagerRef);
QVERIFY(moduleManager != NULL);

// Validar que el componente esté disponible.
QVERIFY(moduleManager->listAviableModules().contains("moonlightDE-iddle_module"));

// Cargar el componente.
moduleManager->load("moonlightDE-iddle_module");

// Validar que el componente fue cargado.
QVERIFY(moduleManager->listActiveModules().contains("moonlightDE-iddle_module"));

// Descargar el componente.
moduleManager->unload("moonlightDE-iddle_module");

// Validar que el componente fue descargado.
QVERIFY(moduleManager->listActiveModules().contains("moonlightDE-iddle_module") ==
false);
```

Ilustración 10: Prueba interfaz IModuleManager

3.2.3 Interfaz *IEnvironment*

La interfaz *IEnvironment* permite que los componentes accedan y modifiquen las variables de entorno de la aplicación. Responde a los requisitos funcionales: notificar cambios en las variables de entorno, gestionar variables de entorno de ejecución, registrar servicios, buscar servicios y utilizar servicios. Su validación se realiza utilizando las instrucciones que se muestran a continuación:

```
// Obtener y validar el contexto del componente.
ModuleContext * context = GetModuleContext();
QVERIFY(context != NULL);

// Obtener el descriptor del servicio.
 ServiceReference<Core::IEnvironment> iEnvironmentRef =
            context->GetServiceReference<Core::IEnvironment>();

// Obtener y validar el servicio.
Core::IEnvironment * iEnvironment = context->GetService(iEnvironmentRef);
QVERIFY(iEnvironment != NULL);

// Conectar la señal “environmentUpdated” al procedimiento “updateValue” para
validar su emisión.
QObject::connect(iEnvironment, SIGNAL(environmentUpdated(QString, QString)),
            this, SLOT(updateValue(QString, Qstring)));

// Insertar una variable con un valor.
iEnvironment->insert("TEST_VALUE", "VALUE");

// Validar que la variable fue insertada.
QCOMPARE(values.value("TEST_VALUE"), Qstring("VALUE"));
QCOMPARE(iEnvironment->generateProcessEnvironment().value("TEST_VALUE"),
Qstring("VALUE"));

// Eliminar la variable.
iEnvironment->remove("TEST_VALUE");

// Validar que la variable fue eliminada.
QVERIFY(iEnvironment->generateProcessEnvironment().value("TEST_VALUE") ==
QString(""));
QVERIFY(values.value("TEST_VALUE") == QString(""));
```

Ilustración 11: Prueba interfaz IEnviroment

3.2.4 Interfaz *ISettingsProfile*

Para acceder a las configuraciones privadas los componentes utilizan la interfaz *ISettingsProfile*. Esta se relaciona con los requisitos funcionales: iniciar utilizando perfiles de configuración, registrar servicios, buscar servicios y utilizar servicios. Su validación se realiza mediante las instrucciones siguientes:

```
// Obtener y validar el contexto del componente.
ModuleContext * context = GetModuleContext();
QVERIFY(context != NULL);

// Obtener el descriptor del servicio.
ServiceReference<Core::ISettingsProfile> iSettingsProfileRef =
            context->GetServiceReference<Core::ISettingsProfile>();

// Obtener y validar el servicio.
Core::ISettingsProfile * settingsProfile = context-
>GetService(iSettingsProfileRef);
QVERIFY(settingsProfile != NULL);

// Solicitar el registro de configuración.
QSettings *settings = settingsProfile->getSettingsOf("TestSettings");

// Insertar un valor en el registro de configuración.
settings->setValue("Key", "Value");

// Cerrar el registro de configuración.
delete(settings);

// Reabrir el registro de configuración.
settings = settingsProfile->getSettingsOf("TestSettings");

// Validar que el valor exista y coincida con el incertado.
QCOMPARE(settings->value("Key").toString(), QString("Value"));
```

Ilustración 12: Prueba interfaz ISettingsProfile

3.3 Validación de los requisitos relacionados al desarrollo de componentes

Con la ejecución de las pruebas antes descritas se consigue validar el funcionamiento del Núcleo, sin embargo aún queda pendiente validar los requisitos relacionados al desarrollo de componentes. Estos requieren el conocimiento de las tecnologías utilizadas en la elaboración del sistema por lo que debe ser probado por desarrolladores.

La validación de los requisitos funcionales crear componentes, modificar componentes y desplegar componentes se realizó mediante el desarrollo de un prototipo de panel para el EE. Esta tarea fue realizada de conjunto con la comunidad de usuarios del proyecto Nova los cuales actuaron como probadores del sistema desarrollado.

Para facilitar esta tarea el código de la aplicación fue colocado en la forja de proyectos GitHub[38]. Se creó una wiki[39] con la documentación del proyecto donde se incluyó una guía para los desarrolladores de

[38] Forja para alojar proyectos de software utilizando el sistema de control de versiones Git.
[39] Sistema de gestión documental que permite la edición de documentos de manera colaborativa utilizando Internet.

componentes. Además se desarrolló una plantilla de componente donde se especifican detalles de la implementación del mismo (ver Anexo 6).

Luego de una semana de desarrollo se obtuvo un prototipo funcional de la aplicación embebida Panel. Esta se conforma por los siguientes componentes (ver Ilustración 13):

- Un panel, encargado de reservar el espacio en pantalla y posicionar los otros componentes funcionales.
- Un menú de aplicaciones.
- Área de tareas donde se muestra el estado de las ventanas.
- Área de lanzadores rápidos.
- Un reloj que muestra la hora, fecha y el calendario.

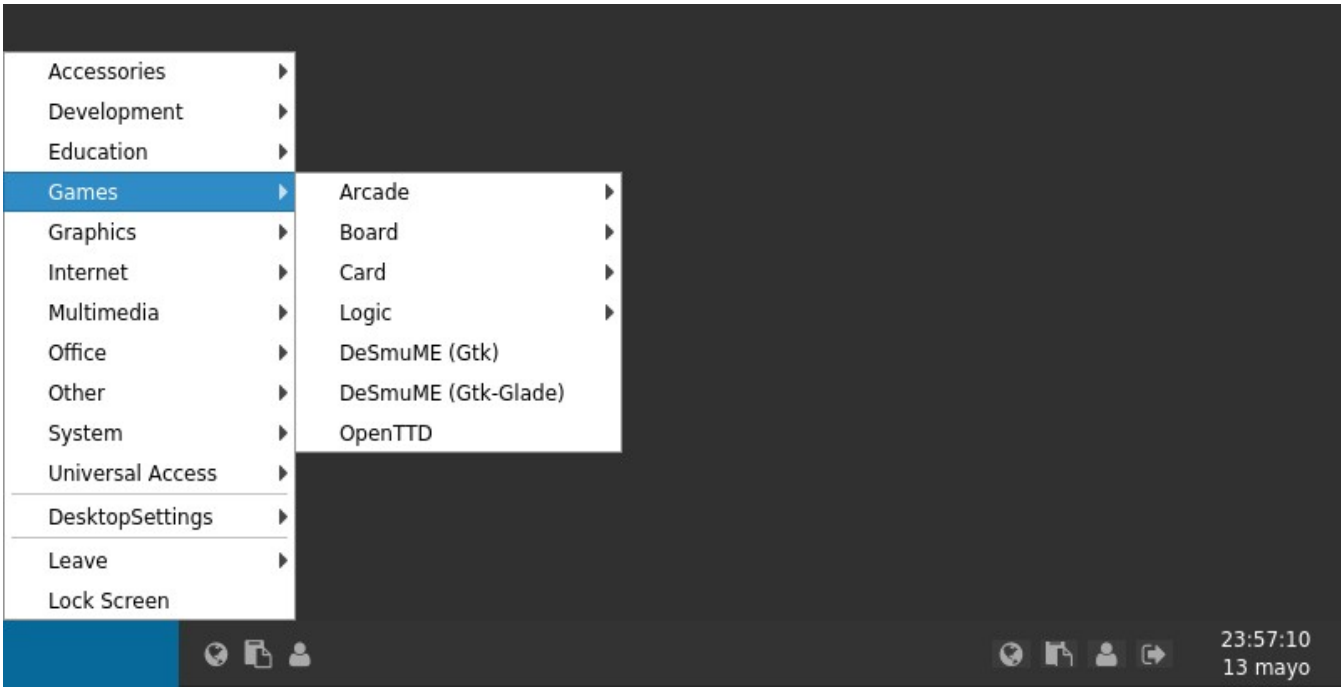

Ilustración 13: Captura de pantalla de la aplicación embebida Panel

3.4 Validación de los requisitos no funcionales

Para la validación de los requisitos no funcionales se realizaron un conjunto de pruebas en equipos con diferentes prestaciones de hardware:

Equipo	Procesador	RAM	Sistema operativo
1	Intel® Core® i3 M 380 a 2.53GHz	2 x SODIMM DDR3 2 GiB[40]	Lubuntu 14.04
2	Intel® Celeron® CPU 2.66GHz	2 x DIMM SDRAM 512 MiB	Lubuntu 14.04
3	Intel® Celeron® CPU 2.93GHz	DIMM SDRAM 256 MiB	Lubuntu 14.04
4	Intel® Celedon® CPU 215 @1.33GHz	DIMM SDRAM 256 MiB	Lubuntu 14.04

Tabla 11: Características de los equipos utilizados en las pruebas

3.4.1 Tiempo de inicio

El tiempo de inicio fue medido utilizando la herramienta “time”, incluida en el sistema. Para ello se ejecutó la aplicación utilizando el perfil de configuración para las pruebas, este realiza la validación de las

[40] **Gibibyte:** unidad de almacenamiento de información equivalente a 2^{30} bytes. Definido en el estándar IEC 80000-13:2008.

interfaces y termina la aplicación. A continuación se muestra el resultado de la ejecución de la prueba en diferentes equipos. Como se puede apreciar en ninguno de los casos el tiempo real excede los 500 ms cumpliéndose con el requisito no funcional establecido.

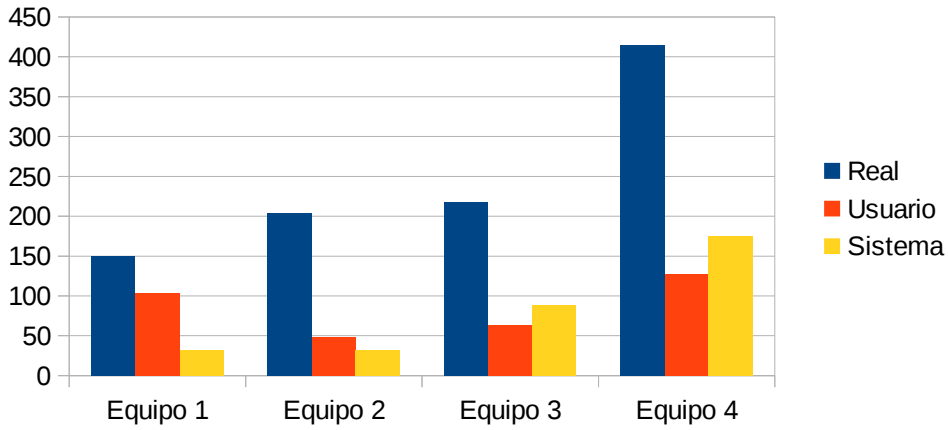

Ilustración 14: Tiempo de inicio del sistema en ms

3.4.2 Consumo de memoria

Medir con exactitud el consumo de memoria de una aplicación constituye una tarea compleja debido al gran número de factores que influyen en el resultado. Una aplicación al ser cargada en memoria reserva espacio para el código del ejecutable, para las bibliotecas compartidas y para la pila[41]. El código de la aplicación y las bibliotecas compartidas son cargadas por el núcleo del sistema operativo cuando este estime conveniente o la aplicación lo solicite. Las bibliotecas compartidas además son cargadas una sola vez en memoria de manera que todas las aplicaciones pueden acceder a las funcionalidades que posee. En contraposición, la pila es manejada directamente por la aplicación y es utilizada para almacenar los datos en tiempo de ejecución por ello puede ser medida con mayor precisión. En consecuencia se decide medir la memoria utilizada en la pila en lugar del total utilizado por la aplicación.

[41] **Pila (heap):** Espacio de memoria que es reservado de manera dinámica por los procesos.

Para realizar esta tarea se utilizó la herramienta “Valgrind”. Esta provee a manera de módulos un conjunto de funcionalidades para el análisis de la memoria, específicamente se utilizó el módulo “massif” permite medir de manera precisa el uso de la pila. En la gráfica que se muestra a continuación (ver Ilustración 15) se muestra el consumo de memoria de la aplicación desde que es ejecutada hasta que termina. En esta se aprecia el aumento de la pila (eje Y) a medida que avanza el tiempo (eje X), al final de la ejecución la aplicación llega a consumir 1904944 bytes (1,816 MiB) de RAM por lo que se afirma que se cumple con el requisito no funcional que establece que la aplicación debe consumir menos de 3 MiB de memoria independientemente de los componentes.

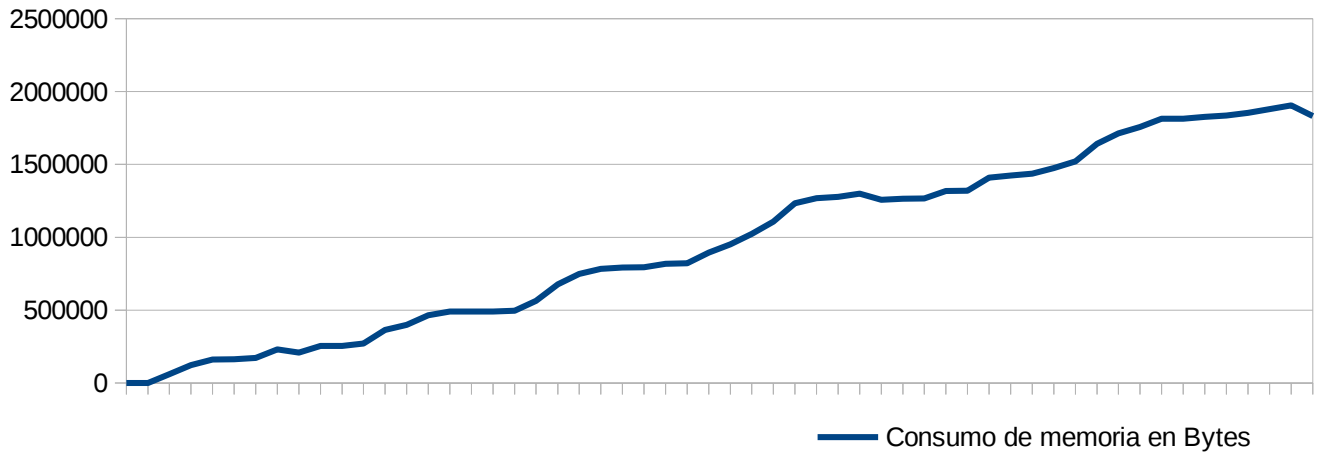

Ilustración 15: Consumo de memoria del Núcleo

3.5 Análisis del impacto de la propuesta de arquitectura en el consumo de recursos del EE.

El Núcleo constituye solo una fracción del EE, por ello para conocer su consumo de recursos total es necesario medir todos sus componentes. Debido a que actualmente estos no se encuentran desarrollados esta tarea resulta imposible. Sin embargo se puede hacer una estimación a partir del análisis de los cambios propuestos en la arquitectura. Para ello se toma como caso de estudio las deficiencias detectadas en el EE.

La primera deficiencia está relacionada con el procedimiento utilizado por la herramienta de configuración de intermediarios (proxy) para compartir datos con otras aplicaciones, datos que ocupan aproximadamente 1 KiB[42]. de RAM. Este se consideró inapropiado porque replicaba los datos y requería el reinicio de sesión para hacer efectivas las configuraciones. Con el desarrollo del Núcleo, los datos de los intermediarios de red podrán ser publicados como servicios poniéndolos a disposición de todos los componentes del EE,

[42] **Kibibyte:** unidad de almacenamiento de información equivalente a 2^{10} bytes. Definido en el estándar IEC 80000-13:2008.

eliminando la necesidad de reiniciar la sesión y la replicación de datos. En este caso el ahorro de recursos por concepto de no replicación de datos es de aproximadamente 21 KiB.

La segunda deficiencia consistía en que se duplicaban las funcionalidades y datos para la gestión de la descripción de las aplicaciones instaladas. Nova Ligero 2013 recién instalado posee 74 aplicaciones de usuario, lo que representan aproximadamente 3111 KiB de texto e iconos. La lista de las aplicaciones instaladas en el sistema se muestra de manera íntegra en tres aplicaciones diferentes (panel, el gestor de ventanas, el gestor de ficheros) y parcialmente en una cuarta (centro de configuraciones) por lo que los autores de este trabajo afirman, que se consume un total de aproximadamente 10 MiB. Con la creación de un único servicio para esta funcionalidad se eliminará la duplicación de código y se ahorrarán aproximadamente 7MiB de RAM.

La ausencia de un único sistema de gestión de atajos de teclado constituye la tercera deficiencia detectada en el EE de Nova Ligero. De manera similar a la situación anterior se incurre en la duplicación de la implementación de la funcionalidad. En este caso se propone exponer la funcionalidad en un servicio para eliminar las colisiones, pero como los atajos de teclado no se repiten no se ahorra RAM.

La cuarta deficiencia consiste en la duplicación del ciclo de ejecución de GTK. Por cada aplicación se consume aproximadamente 1 MiB en el aspecto ante mencionado. En el EE de Nova Ligero 2013 coexisten 22 aplicaciones por ello se invierte un total de 22 MiB de RAM. La arquitectura propuesta reduce la cantidad de aplicaciones independientes a 3 y transforma al resto en aplicaciones embebidas en el Núcleo de manera que compartan el mismo ciclo de ejecución. De esta manera se espera aumentar la disponibilidad de RAM en 19 MiB.

De manera general Nova Ligero 2013 luego de ser instalado consume un total de 90 MiB de RAM. De ser aplicadas las transformaciones antes descritas se podrá reducir este valor en aproximadamente 26, 02 MiB de manera que el sistema consumiría solo 63,98 MiB.

Funcionalidad	Actual (2013)	Esperado (2015)	Diferencia
Configuración de intermediarios	22 KiB	1 KiB	21 KiB
Descripción de aplicaciones	10 MiB	3 MiB	7 KiB
Gestión de atajos de teclado	~ 40 KiB	~ 40 KiB	0 KiB
Ciclo de ejecución de GTK	22 MiB	3 MiB	19 MiB
Nova Ligero	90 MiB	63,98 MiB	26,02 MiB

Tabla 12: Análisis del impacto de la arquitectura propuesta en el consumo de recursos del EE

3.6 Conclusiones del Capítulo

En este capítulo se demostró la conformidad del producto desarrollado con los requisitos funcionales y no funcionales. En este proceso se comprobó que el Núcleo consume aproximadamente 1.8 MiB de RAM e inicia en menos de 500 ms en los equipos probados. Por último se analizó el posible impacto de la arquitectura propuesta en la disponibilidad de recursos del sistema, identificándose la posibilidad de ahorrar aproximadamente 26 MiB de RAM.

Conclusiones Generales

Los resultados obtenidos durante el desarrollo de la presente investigación permiten arribar a las siguientes conclusiones:

- El estudio de tendencias en el campo de la interoperabilidad de sistemas de software arrojó que el estilo arquitectónico orientado a servicios permite lograr la interoperabilidad entre componentes de software.
- El estudio de los mecanismos de comunicación entre componentes de software arrojó que es necesario transformar las aplicaciones del EE en bibliotecas dinámicas que sean cargadas por una misma aplicación para lograr la interoperabilidad sin afectar la disponibilidad de recursos del sistema.
- El desarrollo del Núcleo del EE de Nova Ligero 2015, basado en la arquitectura propuesta, permite la interoperabilidad entre sus componentes sin afectar la disponibilidad de recursos.

Recomendaciones

- Reestructurar las aplicaciones del entorno de escritorio como componentes (bibliotecas dinámicas) para el Núcleo de acuerdo a la arquitectura propuesta.
- Probar el software desarrollado en equipos con diferentes características de hardware.
- Comparar el EE de Nova Ligero 2015 luego de desarrollado según la arquitectura propuesta con su antecesor.
- Exportar los servicios ofrecidos por el EE a productos desarrollados por terceros, mediante la utilización de D-Bus.
- Integrar al entorno de escritorio los servicios estandarizados por la fundación *FreeDesktop*.

Referencias Bibliográficas

- **ALBALAT**, Miguel y RAMIREZ, Barbarita. *Solución Para Hacer De Guano Un Entorno de escritorio usable*. Universidad de las Ciencias Informáticas, 2009. 005.43-Alb-S-TD-2448-09
- A. W. **Brown** and K. C.Wallnau. *The current state of CBSE*. IEEE Software, 1998. 15(5):37-46.
- **Berzal**, Fernando. *Comunicación entre Procesos*. [en línea]. 2009. [Consultado: 6 de diciembre 2013]. Disponible en: http://elvex.ugr.es/decsai/java/pdf/F1-ipc.pdf
- **CALISOFT**. *Calisoft*. [en línea]. 2009. [Consultado: 1 de febrero 2014]. Disponible en: http://calisoft.uci.cu/index.php/servicios
- **Carnegie** Mellon University. *A Framework for Software Product Line Practice, Version 5.0*. [en línea]. 2013. [Consultado: 11 de febrero 2014]. Disponible en: http://www.sei.cmu.edu/productlines/frame_report/softwareSI.htm
- **CABRERA** MOLINA, Daniel. *Tutorial de CMake*. [en línea]. 2009. [Consultado: 17 mayo 2014]. Disponible en: https://labs.isee.biz/images/e/ef/Cmake-material.pdf
- **CERIA**, Santiago. *Casos de Uso -Un método práctico para explorar requerimientos.* [en línea]. 2011. [Consultado: 16 de Mayo 2014]. Disponible en: http://www-2.dc.uba.ar/materias/isoft1/2001_2/apuntes/CasosDeUso.pdf
- **EPF PROJECT**, EPF Wiki. [en linea]. 2014.[Consultado: 3 febrero 2014]. Disponible en: http://epf.eclipse.org/wikis/openupsp/openup_basic/disciplines/test,_0TkKQMlgEdmt3adZL5Dmdw.html
- **Fernández** del Monte, Yusleydi. *Metodología para desarrollar la distribución cubana de GNU/Linux Nova*, 2013.
- **GARCÍA** RIVAS, Dairelys. *Corrección De Errores a Nova Ligero 3,0*. La Habana: Universidad de las Ciencias Informáticas, 2012.005.43-Gar
- **German Cancer Research Center**, CTK Plugin Framework: Introduction – Commontk. [en línea]. 2014. [Consultado: 18 de febrero 2014]. Disponible en: http://www.commontk.org/index.php/Documentation/CTK_Plugin_Framework
- **Git Project**. *Git - Acerca del control de versiones*. [en línea]. 2013. [Consultado: 5 de abril 2014]. Disponible en: http://git-scm.com/book/es/Empezando-Acerca-del-control-de-versiones.

- **Hohpe**, Gregor y Woolf, Bobby. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. s.l.: Addison -Wesley, 2003. 0321200683.
- **HOFF**, Todd, C++ Coding Standard. [en linea]. 2008. [Consultado: 2 febrero 2014]. Disponible en: http://www.possibility.com/Cpp/CppCodingStandard.html
- **ISO**. *ISO Draft International Standard ISO/DIS 19101 Geographic information -Reference Model,* 2001. p. 47.
- **IEEE**. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. [en línea]. 1990. [Consultado: 20 de enero 2014]. Disponible en: https://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=2267
- **IEEE**. *Recommended Practice for Software Requirements Specifications*. [en línea]. 2009. [Consultado: 16 de mayo 2014]. Disponible en: http://www.ieee.org/index.html?WT.mc_id=hpf_logo
- García de **Jalón**, Javier and AGUINAGA, Iker and MORA, Alberto. *Aprenda LINUX como si estuviera en primero*. Universidad de Navarra, San Sebastián, 2000. página 57.
- **KAPTELININ**, Victor y CZERWINSKI, Mary. *Beyond the Desktop Metaphor: Designing Integrated Digital Work Environments* [en línea]. MIT Press Books. The MIT Press, 2007. [Consultado: 22 de mayo 2014]. Disponible en: http://econpapers.repec.org/bookchap/mtptitles/026211304x.htm
- **Koch,** Stefan. *Free/Open Source Software Development. Chicago: IGI Global*. 1-328. Web. 18 Oct. 2011. doi:10.4018/978-1-59140-369-2.
- **KOSKELA**, Mika, RAHIKAINEN, Mikko and WAN, Tao. *Software development methods: SOA vs. CBD, OO and AOP* [en línea]. Helsinki University of Technology, 2007. [Consultado: 3 de marzo 2014]. Disponible en: http://www1.soberit.hut.fi/T-86/T-86.5165/2007/final_koskela_rahikainen_wan
- **KRUCHTEN**, Philippe. *The «4+1» View Model of Software Architecture.* [en línea]. 1995. [Consultado: 22 de abril 2014]. Disponible en: http://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf
- **LARABEL**, Michael. *[Phoronix] Power & Memory Usage Of GNOME, KDE, LXDE & Xfce*. [en línea]. 2010. [Consultado: 3 de mayo 2014]. Disponible en: http://www.phoronix.com/scan.php?page=article&item=linux_desktop_vitals&num=1
- **Linthicum**, David S. *Enterprise Application Integration* s.l.: Addison –Wesley, 1999. 0201615835.
- **LXDE.ORG**. *LXDE*. [en línea]. 2012. [Consultado: 5 de marzo 2014]. Disponible en: http://lxde.org/e

s/lxde

- **MACKENZIE**, David. *Linux Monitorización de rendimiento*. [en línea]. 2010. [Consultado: 2 de junio 2014]. Disponible en: http://www.dsi.uclm.es/personal/AntonioBueno/ESI/monitor%20en%20linux.pdf
- **Marquez** Garcia, Francisco. *UNIX. Programación Avanzada, 2004*. 3ª edición. ISBN 978-84-7897-603-4. Editorial: RA-MA EDITORIAL.
- **MARTINEZ**, Ivet Carolina. *Modelo Conceptual/ Modelo de dominio*. *Universidad Simón Bolívar. [en línea]. 2012.* [Consultado: 20 de marzo 2014]. Disponible en: http://ldc.usb.ve/~martinez/cursos/ci3715/clase6_AJ2010.pdf
- **Mangarelli**, Eduardo y Cabrera, Martín. *SOA Introduction*. Uruguay, 2006: s.n.
- **Marshall**, Dave. *Remote Procedure Calls*. [en línea]. 1999. [Consultado: 15 de enero 2014]. Disponible en:<http://www.cs.cf.ac.uk/Dave/C/node33.html>
- **MENECES** y OSORIA, José. *Guía metodológica de interoperabilidad para sistemas integrales de gestión en Cuba*. [en línea]. 2010. [Consultado: 3 de febrero 2014]. Disponible en: http://repositorio_institucional.uci.cu/jspui/bitstream/ident/TD_03266_10/1/TD_03266_10.pdf
- **Meier** J.D, Hill David, Alex Hommer, and Jason Taylor. *Microsoft Application Architecture Guide*, 2nd Edition. [en línea]. 2009. [Consultado: 10 de noviembre 2010]. Disponible en: http://msdn.microsoft.com/enus/library/dd673617.aspx
- **MITCHELL**, Mark, OLDHAM, Jeffrey and SAMUEL, Alex. *Advanced Linux Programming* [en línea]. Sams Publishing, 2001. [Consultado: 11 de mayo 2014]. ISBN9780735710436. Disponible en: http://www.cse.hcmut.edu.vn/~hungnq/courses/nap/alp.pdf
- **MCLAUGHLIN**, Brett D., POLLICE, Gary and WEST, David. *Object-Oriented Analysis and Design.* [en línea]. 2007. [Consultado: 30 de abril 2014]. Disponible en: http://oreilly.com/catalog/hfobjects/chapter/ch03.pdf
- **OPEN UP**. *Open UP: Metodología Open UP*. [en línea]. 2012. [Consultado: 12 de febrero 2014]. Disponible en: http://openupeaojmp.blogspot.com/2013/09/metodologia-open-up.html
- **OSGI ALLIANCE**. *OSGi Service Platform Core Specification.* [en línea]. 2011. [Consultado: 14 de diciembre 2013]. Disponible en: http://www.osgi.org/download/osgi.residential-4.3.0-pfd.pdf

- **PIERRA** FUENTES, Allan. *Nova, distribución cubana de GNU/Linux : reestructuración estratégica de su proceso de desarrollo*. S.l.: Universidad de las Ciencias Informáticas. [en línea]. 2011. [Consultado: 8 de enero 2013]. Disponible en: http://repositorio_institucional.uci.cu/jspui/bitstream/ident/TM_04841_11/1/TM_04841_11.pdf.005.43-Pie-N-TM04841-11
- **QMLBOOK**. *Qt5 Cadaques—Qt5 Cadaques Book v2014-01*. [en línea]. 2014. [Consultado: 15 de enero 2014]. Disponible en: http://qmlbook.org/index.html
- **QT PROJECT**, Writing a Unit Test. [en linea]. 2013. [Consultado: 18 de enero 2015]. Disponible en: http://qt-project.org/doc/qt-4.8/qtestlib-tutorial1.html
- **RODRÍGUEZ**, Adisleydis y FÍRVIDA DONÉSTEVEZ, Abel. *Guano, Entorno De Escritorio Cubano, Libre*, 2009. S.l.:s.n.005.43-Rod-G-TD-2436-09
- **REYNOSO**, Carlos y KICILLOF, Nicolás. *Estilos y Patrones en la Estrategia de Arquitectura de Microsoft.* [en línea]. 2004. [Consultado: 4 de mayo 2014]. Disponible en: http://carlosreynoso.com.ar/archivos/arquitectura/Estilos.PDF
- **RUBEL**, Dan. *The Heart of Eclipse*. Queue - System Evolution [en línea] Vol. 4, no. 8 (2006), p. 36–44. DOI 10.1145/1165754.1165767. [Consultado: 15 de enero 2014]. Disponible en: http://doi.acm.org/10.1145/1165754.1165767
- **RUMBAUCH**, James, JACOBSON, Ivar and BOOCH, Grady. *El Lenguaje Unificado de Modelado Manual de Referencia*. [en línea]. 2000. [Consultado: 16 de mayo 2014]. Disponible en: http://ingenieriasoftware2011.files.wordpress.com/2011/07/el-lenguaje-unificado-de-modeladomanual-de-referencia.pdf
- **SARRIEGUI**, Jose M., GARCÍA DE JALÓN, Javier, RODRÍGUEZ, José Ignacio y BRAZÁLEZ, Alfonso. *Aprenda C++ como si estuviera en primero.* [en línea]. 1998. [Consultado: 17 de mayo 2014]. Disponible en: http://mat21.etsii.upm.es/ayudainf/aprendainf/Cpp/manualcpp.pdf
- **SWEET**, David. *KDE 2.0 Development.* [en línea]. 2001. [Consultado: 15 de febrero 2014]. Disponible en: ftp://riksun.riken.jp/X11/kde-jp/kde/document/kde2dev.pdf
- **SAN,** JOSÉ A. *Arquitectura Basada en Componentes*. [en línea]. 2013. [Consultado: 19 de febrero 2014]. Disponible en: http://es.scribd.com/doc/14704374/Arquitectura-Basada-en-Componentes
- **THELIN**, Johan, Foundations of Qt development [en linea]. 2007. Berkeley, CA; New York: Apress. [Consultado: 12 de diciembre 2013]. ISBN9781430202516 1430202513 9781590598313

1590598318. Disponible en: http://bvs.sld.cu/revistas/mciego/informatica/data/APress_-_Foundations_of_Qt_Development.pdf

- **VISUAL PARADIGM**. *VISUAL PARADIGM 10.1.* [en línea]. 2013. [Citado el: 19 de febrero de 2014]. Disponible en: http://www.visual-paradigm.com/product/vpuml/
- **VALGRIND DEVELOPERS**. Valgrind Home. [en línea]. 2013. [Consultado: 1 de junio 2014]. Disponible en: http://valgrind.org/
- **Young**, Ralph R. *The Requirements Engineering Handbook.* Norwood ARTECH, 1990.
- **YERPES** BAENA, Óscar. *Estudio comparativo de distribuciones GNU/Linux*. [en línea]. 2012. [Consultado: 17 de mayo 2014]. Disponible en: http://openaccess.uoc.edu/webapps/o2/bitstream/10609/8190/1/oyerpesTFC0611.pdf
- **ZELZER**, Sascha. *Going Native With The OSGi Service Layer*. En "OSGi Community Event", Ludiwgsburg. [en línea]. 2013. [Consultado: 20 de febrero 2014]. Disponible en: http://www.osgi.org/CommunityEvent2013/Speakers#anchor4

Bibliografía Consultada

- ALBALAT, Miguel y RAMIREZ, Barbarita. *Solución Para Hacer De Guano Un Entorno de escritorio usable*. Universidad de las Ciencias Informáticas, 2009. 005.43-Alb-S-TD-2448-09
- SAN, JOSÉ A. *Arquitectura Basada en Componentes*. [en línea]. 2013. [Consultado: 19 de febrero 2014]. Disponible en: http://es.scribd.com/doc/14704374/Arquitectura-Basada-en-Componentes
- *Comunicando aplicaciones con D-Bus* (*Parte I)* | *humanOS*. [en línea]. [Consultado: 21 de marzo 2014]. Disponible en: http://humanos.uci.cu/2013/11/humancode-comunicando-aplicaciones-con-d-bus-parte-i/
- *D-Bus*. [en línea]. [Consultado: 1 de febrero 2014]. Disponible en: http://es.wikipedia.org/wiki/D-Bus
- *Desktop metaphor*. [en línea]. 2012. [Consultado: 9 de abril 2014]. Disponible en: http://en.wikipedia.org/wiki/Desktop_metaphor
- EcuRed. *Lenguaje de Modelado Unificado*. [en línea]. 2012. [Consultado: 3 de febrero 2014]. Disponible en: http://www.ecured.cu/index.php/UML
- *Integración de Aplicaciones*. [en línea]. 2011. [Consultado: 4 de febrero 2014]. Disponible en: http://www.softwareag.com/latam/solution/application_integration/default.asp
- *Integración de dispositivos y aplicaciones de terceros mediante D-Bus | MCP*. [en línea]. [Consultado: 1 de febrero 2014]. Disponible en: http://www.vision-robotics.com/integracion-de-dispositivos-y-aplicaciones-de-terceros-mediante-d-bus
- RODRÍGUEZ, Adisleydis y FÍRVIDA DONÉSTEVEZ, Abel. *Guano, Entorno De Escritorio Cubano, Libre*, 2009. S.l.:s.n.005.43-Rod-G-TD-2436-09
- KOSKELA, Mika, RAHIKAINEN, Mikko and WAN, Tao. *Software development methods: SOA vs. CBD, OO and AOP* [en línea]. Helsinki University of Technology, 2007. [Consultado: 3 de marzo 2014]. Disponible en: http://www1.soberit.hut.fi/T-86/T-86.5165/2007/final_koskela_rahikainen_wan
- MENECES y OSORIA, José. *Guía metodológica de interoperabilidad para sistemas integrales de gestión en Cuba*. [en línea]. 2010. [Consultado: 3 de febrero 2014]. Disponible en: http://repositorio_institucional.uci.cu/jspui/bitstream/ident/TD_03266_10/1/TD_03266_10.pdf

Glosario de Términos

Distribución: Distribución de software basada en el núcleo Linux que incluye determinados paquetes de software para satisfacer las necesidades de un grupo específico de usuarios, dando así origen a ediciones domésticas, empresariales y para servidores.

Gestor de ficheros: Aplicación informática que provee acceso a archivos y facilita el realizar operaciones con ellos, como copiar, mover, eliminar archivos donde el usuario lo quiera ubicar y poder ingresar a ellos para realizar ciertas tareas.

Gestor de sesiones: Aplicación encargada de salir de la sesión, hibernar, suspender, cambiar de usuario, apagar y reiniciar.

Gestor de ventanas: Es un programa que controla la ubicación y apariencia de las aplicaciones bajo el Sistema de Ventanas X[43], así como los múltiples espacios de trabajo virtuales.

Lanzador de aplicaciones: Aplicación encargada de suministrar un menú de aplicaciones, un espacio para el reloj, los iconos de notificaciones, accesos directos a aplicaciones, el paginado de escritorio, applets para el control de volumen, el acceso a red y una barra de tareas donde se sitúan las aplicaciones minimizadas.

Aplicaciones de terceros: Son las aplicaciones desarrolladas por proyectos ajenos.

Sistema base: Conjunto de aplicaciones que brindan las funcionalidades mínimas para la explotación de un ordenador, usualmente comprende al sistema operativo y una interfaz de línea de comandos.

[43] El sistema de ventanas X (en inglés X Window System) fue desarrollado para dotar de una interfaz gráfica a los sistemas Unix.

Anexos

Anexo 1: Descripción del caso de uso Implementar componente.

Objetivo	Implementar componente	
Actores	Desarrollador de componentes.	
Resumen	Se crea un nuevo componente a partir de una platilla predefinida.	
Complejidad	Media	
Prioridad	Secundario	
Precondiciones	Se posee la plantilla de un componente.	
Postcondiciones	Se crea un nuevo componente.	
Flujo de eventos		
Flujo básico “Crear componente”		
	Actor	**Sistema**
1.	Copia la plantilla del componente a una nueva ubicación.	
2.	Define el nombre y la versión del componente.	
3.	Implementa los métodos “load” y “unload” del componente.	
4.	Solicita la construcción del componente.	
5.		Compila y enlaza el componente, generando una biblioteca compartida.
6.		Termina el caso de uso

Anexo 2: Descripción del caso de uso Modificar componente.

Objetivo	Modificar componente	
Actores	Desarrollador de componentes.	
Resumen	Se modifica un componente existente y se aumenta su versión.	
Complejidad	Media	
Prioridad	Secundario	
Precondiciones	Se posee el código fuente del componente.	
Postcondiciones	Se crea una nueva versión del componente.	
Flujo de eventos		
Flujo básico “Modificar componente”		
	Actor	**Sistema**
1.	Incrementa la versión del componente.	
2.	Modifica el código del componente.	
3.	Solicita la construcción del componente.	
4.		Compila y enlaza el componente, generando una biblioteca compartida.
5.		Termina el caso de uso

Anexo 3: Descripción del caso de uso Desplegar componente.

Objetivo	Desplegar componente

Actores	Desarrollador de componentes.	
Resumen	Se instala la biblioteca compartida y sus recursos.	
Complejidad	Media	
Prioridad	Secundario	
Precondiciones	Se posee la biblioteca compartida del componente.	
Postcondiciones	La biblioteca compartida y sus recursos son instalados.	
Flujo de eventos		
Flujo básico “Desplegar componente”		
	Actor	**Sistema**
1.	Define los recursos del componente.	
2.	Solicita el despliegue del componente.	
3.		Instala en el sistema la biblioteca compartida y sus recursos.
4.		Termina el caso de uso

Anexo 4: Descripción del caso de uso Configurar perfil.

Objetivo	Configurar perfil	
Actores	Integrador del escritorio	
Resumen	Define los componentes que serán cargados por el Núcleo.	
Complejidad	Baja	
Prioridad	Secundario	
Precondiciones	El Núcleo y los componentes deben estar desplegados en el sistema.	
Postcondiciones	Se genera una lista de los componentes que serán cargados por el Núcleo.	
Flujo de eventos		
Flujo básico “Configurar perfil”		
	Actor	**Sistema**
1.	Ejecuta el sistema con el argumento “setup”.	
2.		Muestra los componentes los desplegados.
3.	Selecciona los componentes que deben ser cargados.	
4.		Actualiza el archivo del perfil de configuración.
5.		Termina el caso de uso.

Anexo 5: Descripción del caso de uso Gestionar variable de entorno.

Objetivo	Gestionar variable de entorno	
Actores	Componente	
Resumen	Se crean, modifican o eliminan las variables de entorno.	
Complejidad	Media	
Prioridad	Secundario	
Precondiciones		
Postcondiciones		
Flujo de eventos		
Flujo básico “Crear variable de entorno”		
	Actor	**Sistema**
1.	Define el nombre y el valor de la variable de entorno.	

2.	Solicita la publicación de la variable.	
3.		Almacena la nueva variable.
4.		Notifica el cambio en el entorno al resto de los componentes.
5.		Termina el caso de uso.
Flujos alternos		
1. “La variable ya está registrada”		
	Actor	**Sistema**
1.	Define el nombre y el valor de la variable de entorno.	
2.	Solicita la publicación de la variable.	
3.		Reemplaza el valor de la variable.
4.		Notifica el cambio en el entorno al resto de los componentes.
5.		Termina el caso de uso.
Sección 1: “Modificar variable de entorno”		
Flujo básico “Modificar variable de entorno”		
	Actor	**Sistema**
1.	Define el nombre y el valor de la variable de entorno.	
2.	Solicita la modificación de la variable.	
3.		Reemplaza el valor de la variable.
4.		Notifica el cambio en el entorno al resto de los componentes.
5.		Termina el caso de uso.
Flujos alternos		
1. “La variable no está registrada”		
	Actor	**Sistema**
1.	Define el nombre y el valor de la variable de entorno.	
2.	Solicita la modificación de la variable.	
3.		Almacena la nueva variable.
4.		Notifica el cambio en el entorno al resto de los componentes.
		Termina el caso de uso.
Sección 2: “Eliminar variable de entorno”		
Flujo básico “Eliminar variable de entorno”		
	Actor	**Sistema**
1.	Solicita que sea eliminada una variable de entorno especificando su nombre.	
2.		Elimina la variable y su valor.
3.		Notifica el cambio en el entorno al resto de los componentes.
4.		Termina el caso de uso.
Flujos alternos		
1. “La variable no está registrada”		
	Actor	**Sistema**
1.	Solicita que sea eliminada una variable de entorno especificando su nombre.	
2.		No se realiza ninguna acción.
3.		Termina el caso de uso.

Anexo 6: Plantilla para el desarrollo de componentes

La plantilla consta de 2 ficheros: CMakeLists.txt y Activator.cpp. Se recomienda utilizar la estructura de carpetas:

- ./CmakeLists.txt
- ./c++/Activator.cpp

```
## ----------------------------------------------------------------
## Build Rules
## ----------------------------------------------------------------

usFunctionGenerateModuleInit( ${PROJECT_NAME}_SOURCES
        NAME ${MODULE_FULL_NAME}
        LIBRARY_NAME "${MODULE_NAME}"
)

add_library(${PROJECT_NAME} SHARED
    ${${PROJECT_NAME}_HEADERS}
    ${${PROJECT_NAME}_SOURCES}
    ${${PROJEC_NAME}_UI_HEADERS}
    ${${PROJEC_NAME}_QRC_SOURCES}
)

qt5_use_modules(${PROJECT_NAME} Core)
qt5_use_modules(${PROJECT_NAME} Gui)
qt5_use_modules(${PROJECT_NAME} Widgets)

# Link the CppMicroServices library
target_link_libraries(${PROJECT_NAME} ${CppMicroServices_LIBRARIES} )

## ----------------------------------------------------------------
## Install
## ----------------------------------------------------------------
INSTALL(TARGETS ${PROJECT_NAME}
    RUNTIME DESTINATION bin
    LIBRARY DESTINATION MODULES_OUTPUT_DIR
    ARCHIVE DESTINATION lib/static)

INSTALL(FILES ${${PROJECT_NAME}_PUBLIC_HEADERS} DESTINATION include/$
{MoonLightDEPrefix} )
```

```
set(MODULE_NAME example)
set(MODULE_FULL_NAME "Example module")
project(moonlightDE${MODULE_NAME})

## ----------------------------------------------------------------
## Sources
```

```
## ------------------------------------------------------------
set(${PROJECT_NAME}_HEADERS
)
set(${PROJECT_NAME}_PUBLIC_HEADERS
)

set(${PROJECT_NAME}_SOURCES
    c++/Activator.cpp
)

set ( ${PROJEC_NAME}_UIS
)

set ( ${PROJEC_NAME}_RESOURCES
)

## ------------------------------------------------------------
## Dependencies - Qt
## ------------------------------- ------------------------------
find_package(Qt5Core    REQUIRED)
find_package(Qt5Gui     REQUIRED)
find_package(Qt5Widgets REQUIRED)

set(CMAKE_AUTOMOC ON)
set(CMAKE_INCLUDE_CURRENT_DIR ON)

QT5_WRAP_UI(${PROJEC_NAME}_UI_HEADERS ${${PROJEC_NAME}_UIS})
QT5_ADD_RESOURCES(${PROJEC_NAME}_QRC_SOURCES ${$
{PROJEC_NAME}_RESOURCES})

## ------------------------------------------------------------
## Dependencies - internal
## ------------------------------------------------------------
include_directories(${INTERFACES_DIR}/${MODULE_NAME})

#CppMicroServices
find_package(CppMicroServices NO_MODULE REQUIRED)
include_directories(${CppMicroServices_INCLUDE_DIRS})

include_directories(${CMAKE_BINARY_DIR})
```

```
#include <usModuleActivator.h>
#include <usModuleContext.h>

US_USE_NAMESPACE

class Activator : public ModuleActivator {
private:

    /**
     * Implements ModuleActivator::Load().
     *
```

```
     * @param context the framework context for the module.
     */
    void Load(ModuleContext* context) {
        std::cout << "Hello!" << std::endl;
    }

    /**
     * Implements ModuleActivator::Unload().
     *
     * @param context the framework context for the module.
     */
    void Unload(ModuleContext* context) {
        std::cout << "Good bye." << std::endl;
    }

};
US_EXPORT_MODULE_ACTIVATOR(example, Activator)
```