Universidad de las Ciencias Informáticas

Facultad 5

ARQUITECTURA DE SOFTWARE PARA EL PROYECTO VISMEDIC.

Trabajo de diploma para optar por el título de

Ingeniero en Ciencias Informáticas

Autor: Alina Dolores Rodríguez Peña

Tutor: M.Sc. Osvaldo Pereira Barzaga

Co-Tutor: Ing. Rubén Alcolea Núñez

Mayo 2013

DECLARACIÓN DE AUTORÍA

Declaramos ser autores de la presente tesis y reconocemos a la Universidad de las Ciencias Informáticas los derechos patrimoniales de la misma, con carácter exclusivo.

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

_____

Alina Dolores Rodríguez Peña

Autor

_____ _____

M.Sc. Osvaldo Pereira Barzaga

Tutor

Ing. Rubén Alcolea Núñez

Co-tutor

DATOS DE CONTACTO

Tutor: M.Sc. Osvaldo Pereira Barzaga.

Edad: 28.

Ciudadanía: Cubano.

Institución: Instituto de Cibernética, Matemática y Física (ICIMAF).

Título: Máster en Informática Aplicada.

Categoría Docente: Instructor.

E-mail: opereira@icimaf.cu

Graduado como Ingeniero en Ciencias Informáticas en la UCI en el año 2008. Tiene ocho años de experiencia en los temas de visualización médica y procesamiento digital de imágenes. Líder del proyecto Simulador Quirúrgico (2008 – 2009). Líder del área temática de Visualización Científica y del proyecto Vismedic (2009 – 2011). Se desempeñó además en el año 2012 como Arquitecto general del software GALBA-CAD, desarrollado por el proyecto CDSEM. Actualmente es miembro del grupo de procesamiento de imágenes digitales del departamento de Matemática Disciplinaria en el ICIMAF.

Co-tutor: Ing. Rubén Alcolea Núñez.

Edad: 26.

Ciudadanía: Cubano.

Institución: Universidad de las Ciencias Informáticas (UCI).

Título: Ingeniero en Ciencias Informáticas.

Categoría Docente: –

E-mail: ralcolea@uci.cu.

Graduado como Ingeniero en Ciencias Informáticas en la UCI en el año 2011. Tiene cuatro años de experiencias en el procesamiento digital de imágenes y la visualización médica. Actualmente es arquitecto del proyecto Vismedic, perteneciente a la Línea Visualización Científica del centro CEDIN. Es miembro del Colectivo de Entrenadores del Movimiento ACM-ICPC en la UCI.

DEDICATORIA

A mis padres, Tania y Juan:

Por tanto cariño, amor y dedicación. Por indicarme siempre el camino que debía seguir y confiar tanto en mí. En especial a mi mamá, por ser este más que mi sueño, el suyo.

A mis hermanitos, Rafael Carlos y Alejandro:

Por ser mi mayor alegría. Solo lamento no haber podido disfrutar sus primeros años tanto como quería, no estar presente cuando dieron el primer paso o pronunciaron la primera palabra. Espero que la vida me permita reparar mi ausencia y que cuando crezcan se sientan orgullosos de su "tati".

A Luis Guillermo:

Por estar presente cada segundo, por cada "regaño" oportuno y por llenar mi vida con tanto amor.

AGRADECIMIENTOS

Quisiera agradecer a todas las personas que de una forma u otra han contribuido a mi formación personal y profesional. A todos los amigos, compañeros y profesores que tanto me han ayudado y enseñado en estos cinco años. Quisiera poder expresar con palabras todo lo que siento y lo mucho que les agradezco, pero en muchas ocasiones las palabras, simplemente, no son suficientes.

De forma especial quiero agradecer:

***A mi papá y a mi mamá**, gracias por apoyarme en todo momento, por hacer de mí una mejor persona y enseñarme a entregarlo todo en cada cosa que hago. En especial a mi mamá, solo escucharte cada día me hizo superar tu ausencia, tus ánimos para seguir me hicieron llegar hasta aquí.*

***A mis hermanitos**, por ser mi impulso.*

***A mis abuelas, mi tía Olga y mi abuelo**, por quererme, consentirme y estar presentes en todo momento.*

***A mis tías y tíos, mis primos y primas**, para mí han sido mucho más que eso, a todos los quiero como a mis padres y a mis hermanos, gracias por todo su cariño y preocupación constante.*

***A Luis Guillermo**, por el amor de cada día, por ayudarme en todo momento, por tu entrega y dedicación, por estar siempre ahí para mí y por toda la magia que sin saber me has regalado.*

***A mis tutores Rubén y Osvaldo**, por su preocupación y apoyo, por haber sido excelentes amigos. Gracias Osval, por cada consejo, por enseñarme tanto y por tu exigencia desde segundo año. "Ruben" tú me adoptaste como "hija", gracias por tu ayuda, por tus sugerencias tan oportunas y por estar presente siempre que te necesité.*

***A Arletis, Dianelis y Yamilka**, gracias por ser amigas incondicionales, por enseñarme el verdadero valor de la amistad, que no requiere de estar cerca constantemente para saber que la otra persona va a estar ahí cuando la necesites. Gracias por cada sueño compartido, cada risa y cada lágrima, por hacerme ver la realidad tantas veces, entenderme y ayudarme siempre, simplemente gracias por ser las hermanas que no tengo.*

***A las niñas del 92105**, a las que están y a las que ya no. En estos cinco años ustedes han sido para mí mucho más que compañeras de apartamento, han sido mi familia. En especial a **Arianna, Lilibet, Lislien y Suly**, por cada momento compartido, por tantos recuerdos lindos, cada una me enseñó algo, gracias por haber confiado en mí, por estar siempre que las necesité, por aterrizarme tantas veces y por ser esas amigas geniales con las que he podido contar.*

***A mis compañeros de aula desde primero hasta quinto**, por cada travesura hecha y por lo mucho que aprendí de ustedes. Especialmente a **Julio y Andy**, por convertirse en verdaderos amigos, por enseñarme y ayudarme tanto, por haberme hecho parte de ese equipo tan lindo que hacemos junto a Yami, a los dos los quiero y admiro muchísimo.*

***A Yadira**, por el deseo de ayudarme siempre, por aceptarme en el colectivo de ISW y por todo lo que me has enseñado.*

***A Ernesto**, por ser Robin Hood tantas veces, gracias por todo.*

***A esta Isla**, por el orgullo de ser cubana, a **Fidel** y la **Revolución**.*

A todos, GRACIAS.

RESUMEN

En los últimos años la Arquitectura de Software se ha consolidado como una disciplina que intenta contrarrestar los efectos negativos que pueden surgir durante el desarrollo de un software, ocupando un rol significativo en la estrategia de negocio de una organización que basa sus operaciones en el software.

En el presente trabajo se propone una arquitectura de software basada en la integración de los estilos arquitectónicos: Arquitectura basada en componentes, Arquitectura basada en capas y Tuberías y filtros, para el proyecto Vismedic, con el objetivo de reducir los problemas de extensibilidad, reusabilidad y dependencia que existían en la arquitectura anterior.

Para realizar la propuesta se hizo necesario el estudio de los conceptos relacionados con la Arquitectura de Software, las características arquitectónicas de tres productos establecidos en el campo del procesamiento y visualización de imágenes (*Volume Rendering Engine* (Voreen), *Visualization Toolkit* (VTK) e *Insight Toolkit* (ITK) y de la especificación OSGi para el desarrollo basado en componentes.

La arquitectura propuesta integra las principales características de las bibliotecas antes mencionadas e incorpora el empleo de *plugins* para extender las funcionalidades. La misma se validó a través de la técnica de evaluación basada en prototipos y de la aplicación del Método de Análisis de Acuerdos de Arquitectura de Software (ATAM). La evaluación permitió identificar los riesgos presentes en la propuesta realizada y determinar que la arquitectura satisface los atributos de calidad definidos para la presente investigación.

Palabras clave: arquitectura de software, componentes, estilo arquitectónico, *plugins*, Vismedic.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

INTRODUCCIÓN

Las organizaciones actuales, independientemente de su tipo, requieren constantemente de aplicaciones informáticas que contribuyan a solucionar problemas relacionados con su proceso de negocio, automatizar las tareas que realizan frecuentemente y que proporcionen asistencia a la hora de tomar decisiones. Para esto se hace necesaria la construcción de eficientes sistemas de software, estos generalmente requieren la combinación de diferentes tecnologías y plataformas de hardware y software para alcanzar un comportamiento acorde con las necesidades de las organizaciones. Los sistemas de software deben ofrecer un alto nivel de rendimiento, adaptarse a las necesidades específicas de la organización y permitir la adición de nuevas funcionalidades con el menor esfuerzo posible. La consecución de estas características exige a los profesionales dedicados al desarrollo de software poner especial atención y cuidado en el diseño de la arquitectura que soportará el funcionamiento del sistema.

Cuba no está exenta a las exigencias que impone el mercado mundial, por tanto, para insertar la Industria Cubana del Software (ICSW) en este mercado, la calidad de los productos que se desarrollan debe ser un elemento fundamental para asegurar la competitividad en un mercado donde, dentro de los principales productores y consumidores de sistemas informáticos, se encuentran los países desarrollados.

Para formar profesionales altamente calificados en la rama de la informática y servir de soporte a la ICSW, surge la Universidad de las Ciencias Informáticas (UCI). Esta institución tiene entre sus objetivos fundamentales desarrollar software para la exportación y el consumo nacional. La UCI se encuentra dividida en centros de desarrollo que se especializan en determinadas áreas o líneas de investigación y desarrollo de software. Tal es el caso del Centro de Informática Industrial (CEDIN) perteneciente a la Facultad 5, este centro se encarga de desarrollar productos y servicios informáticos de automatización industrial y computación gráfica.

Entre los proyectos de desarrollo de software del Departamento de Visualización y Realidad Virtual, perteneciente al centro CEDIN, se encuentra el proyecto de visualización médica Vismedic, este tiene como objetivo, desarrollar aplicaciones que sirvan de apoyo a los médicos en el proceso de diagnóstico a través de imágenes médicas digitales. Actualmente el proyecto cuenta con un Visualizador 2D de imágenes médicas digitales en formato DICOM y se encuentra en la fase de pruebas del Visualizador 3D.

Luego de tres años de desarrollo, se comprobó que al adicionar nuevos requisitos funcionales o cambiar sustancialmente los existentes, se necesita realizar modificaciones en la mayor parte del software. Estas modificaciones se deben al fuerte acople, alto nivel de dependencia entre los elementos que componen el sistema y la débil reusabilidad que se puede alcanzar con los mismos. Las deficiencias presentes en la arquitectura actual ralentizan la incorporación de nuevas características a los productos que se desarrollan en el proyecto, lo que conlleva a notables atrasos en el cronograma de desarrollo y aumenta la curva de aprendizaje para los

nuevos desarrolladores que se incorporan cada curso.

Teniendo en cuenta la situación problémica anterior, se plantea el siguiente **problema científico:** ¿Cómo reducir los problemas de extensibilidad, reusabilidad y dependencia para adicionar nuevas funcionalidades al proyecto Vismedic? A partir del problema planteado, se toma como **objeto de estudio** la arquitectura de software y dentro de esta amplia área se propone como **campo de acción** la arquitectura de software para proyectos de visualización médica.

Se define como **objetivo general** definir y validar una arquitectura de software, que permita reducir los problemas de extensibilidad, reusabilidad y dependencias que presenta la arquitectura actual del proyecto Vismedic. Para dar cumplimiento al objetivo planteado es necesario realizar un grupo de **tareas investigativas** tales como:

1. Elaboración del marco teórico a partir del estado del arte actual referente al tema.
2. Análisis de la arquitectura actual de Vismedic para identificar los cambios necesarios.
3. Análisis de la arquitectura de software de productos establecidos en el campo de la visualización médica, para identificar elementos a reutilizar.
4. Análisis de la especificación OSGi para el desarrollo de software basado en componentes.
5. Diseño de la arquitectura de software para el proyecto Vismedic.
6. Implementación de la arquitectura propuesta.
7. Validación de la propuesta arquitectónica.

Para la realización de la investigación y elaboración del presente trabajo se utilizarán varios **métodos científicos de investigación**, entre los cuales se pueden mencionar:

Métodos teóricos:

- **Histórico – Lógico:** permitirá analizar la evolución y las tendencias actuales de desarrollo de las arquitecturas de software, así como los elementos más importantes de la misma.
- **Analítico – Sintético:** se utilizará para analizar la información sobre las arquitecturas de software para proyectos de visualización médica.
- **Modelación**: se empleará para realizar una representación de la propuesta de arquitectura a utilizar, a través de los diagramas representativos de las vistas arquitectónicas del sistema.
- **Inductivo-deductivo:** permitirá arribar a conclusiones mediante la generalización del conocimiento adquirido desde el análisis.

Métodos empíricos:

- **Pruebas:** se realizarán diferentes pruebas a la arquitectura de software propuesta para determinar si se comporta según los resultados esperados.
- **Observación:** se utilizará para apreciar los resultados obtenidos en la identificación y caracterización de las arquitecturas de software de sistemas similares, con la perspectiva de valorar su factibilidad para mitigar los problemas existentes en el proyecto Vismedic.

- **Consulta de fuentes de información:** se utilizará en la consulta de fuentes bibliográficas para la actualización del conocimiento sobre el tema.

A continuación se muestra la estructura del presente trabajo, incluyendo una síntesis de los capítulos y secciones fundamentales:

Capítulo 1. Fundamentación teórica.

En este capítulo se definen los principales conceptos que serán empleados en el trabajo y se presentan las bases teóricas fundamentales relacionadas con la arquitectura de software. Se muestra una síntesis de los principales estilos y patrones arquitectónicos y de diseño. Se realiza una caracterización de la arquitectura de software de productos similares y de la especificación OSGi para el desarrollo basado en componentes.

Capítulo 2. Solución propuesta.

Este capítulo contiene el diseño de la propuesta de solución, esta se describe utilizando el modelo “4+1” Vistas de la Arquitectura de Software. Se define la metodología de software y las herramientas que se emplearán para el diseño y evaluación de la arquitectura propuesta.

Capítulo 3. Evaluación de la arquitectura.

Este capítulo expone algunos conceptos asociados a la evaluación de las arquitecturas de software. Se evalúa la arquitectura propuesta empleando la técnica basada en prototipos y el Método de Análisis de Acuerdos de Arquitectura de Software.

CAPÍTULO 1. FUNDAMENTACIÓN TEÓRICA

En este capítulo se definen los principales conceptos que serán empleados en el trabajo y se presentan las bases teóricas fundamentales relacionadas con la arquitectura de software. Se aclaran las definiciones de estilos y patrones arquitectónicos, estableciendo las principales diferencias entre ambos términos, se profundiza en los objetivos, ventajas y desventajas de algunos estilos arquitectónicos. Se analizan además los patrones de diseño y de asignación general de responsabilidades y se muestra una síntesis de las principales características de algunos de estos patrones. Se realiza una caracterización de la arquitectura de software de los productos *Volume Rendering Engine, Visualization Toolkit* e *Insight Toolkit* y de la especificación OSGi para el desarrollo basado en componentes.

1.1 Arquitectura de software

Aunque el término arquitectura de software tal como se conoce hoy, aparece en 1992 con el trabajo de Dewayne E. Perry y Alexander L. Wolf: *Foundations for the Study of Software Architecture* [1]*,* sus antecedentes se remontan al menos hasta finales de la década de los sesenta. Actualmente no existe una definición única para el concepto de arquitectura de software, el término ha sido abordado por un gran número de autores, ejemplo de ello lo constituye la colección de definiciones que se encuentra en el sitio oficial del Instituto de Ingeniería de Software (SEI) [2].

Una de las definiciones más reconocidas para el término se encuentra en el estándar 1471 de la IEEE[1]: “La arquitectura de software es la organización fundamental de un sistema enmarcada en sus componentes, las relaciones entre ellos, y el ambiente, y los principios que orientan su diseño y evolución” [3].

Otras definiciones reconocidas internacionalmente para el término son las siguientes:

- “La arquitectura de software, tiene que ver con el diseño y la implementación de estructuras de software de alto nivel. Es el resultado de ensamblar un cierto número de elementos arquitectónicos de forma adecuada para satisfacer la mayor funcionalidad y requerimientos de desempeño de un sistema, así como requerimientos no funcionales, como la confiabilidad, escalabilidad, portabilidad, y disponibilidad” [4].
- “La arquitectura comprende un conjunto de decisiones significativas con respecto a la organización de un sistema de software, incluyendo la selección de los elementos estructurales y sus interfaces, el comportamiento y las especificaciones de la colaboración entre esos componentes, la interdependencia entre los elementos estructurales y de comportamiento en sistemas de gran envergadura y el estilo arquitectónico que guía esta organización” [5].

[1] IEEE, del inglés *Institute of Electrical and Electronics Engineers* (Instituto de Ingenieros Eléctricos y Electrónicos).

- “La arquitectura de software de un sistema es el conjunto de las estructuras necesarias para comprender el mismo, se compone de elementos de software, las relaciones entre ellos y las propiedades de ambos.” [6].

Después del análisis realizado sobre las principales definiciones que existen sobre arquitectura de software, se utilizará para la presente investigación el concepto dado por la IEEE Std 1471-2000, a este concepto se le adicionará que: “La arquitectura debe ser flexible, para permitir la incorporación de nuevas funcionalidades al sistema y soportar además la reutilización de componentes”.

A la hora de diseñar una arquitectura de software se crean y representan componentes que interactúan entre sí, con responsabilidades específicas, y se organizan de forma tal que se logren los requerimientos establecidos. Se puede partir con patrones de soluciones probados que se conocen con el nombre de estilos arquitectónicos, patrones arquitectónicos y patrones de diseño.

1.2 Estilos y patrones arquitectónicos

El proceso de definición de estilo arquitectónico, al igual que el de arquitectura de software, tuvo su inicio con el artículo escrito por Perry y Wolf en 1992, en el mismo plantean: “La década de 1990, creemos, será la década de la arquitectura de software. Usamos el término “arquitectura”, en contraste con “diseño”, para evocar nociones de codificación, de abstracción, de estándares, de entrenamiento formal (de arquitectos de software) y de estilo” [1].

Luego, definen un estilo arquitectónico como una abstracción de tipos de elementos y aspectos formales a partir de diversas arquitecturas específicas. Un estilo arquitectónico encapsula decisiones esenciales sobre los elementos arquitectónicos y enfatiza restricciones importantes de los elementos y sus posibles relaciones [1].

En contraste, Mary Shaw y Paul Clements definen los estilos arquitectónicos como un conjunto de reglas de diseño que identifican los tipos de componentes y conectores que pueden utilizarse para componer un sistema o subsistema, junto con las restricciones locales o globales de la manera en que esta composición se realiza [7].

Roy Thomas Fielding sintetiza la definición de estilo arquitectónico diciendo que: “un estilo arquitectónico es un conjunto coordinado de restricciones arquitectónicas que restringe las funciones/características de los elementos arquitectónicos y las relaciones permitidas entre esos elementos dentro de cualquier arquitectura que se adapte a ese estilo” [8].

En resumen, cada uno de los autores identifican los estilos arquitectónicos partiendo de un denominador común, el alto nivel de abstracción en que se desarrollan y como este permite definir una estructura genérica para la organización de los elementos arquitectónicos que componen el sistema, esta organización implica definir las restricciones de los elementos y sus relaciones.

Además de los estilos también existen los patrones arquitectónicos. Cada patrón describe un

problema que ha ocurrido en reiteradas ocasiones y el núcleo de su solución, de forma que pueda emplearse en múltiples oportunidades sin repetir los esfuerzos iniciales.

Un patrón arquitectónico expresa una organización estructural para los sistemas de software, proporciona un conjunto de subsistemas predefinidos, especifica sus responsabilidades e incluye reglas y directrices para organizar la relación entre ellos. Los patrones arquitectónicos son plantillas para arquitecturas de software concretas. En ellos se especifican las propiedades estructurales de todo el sistema de una aplicación, y tienen un impacto sobre la arquitectura de sus subsistemas. La selección de un patrón arquitectónico es por lo tanto, una decisión de diseño fundamental en el desarrollo de un sistema de software [9].

Es apreciable que en las definiciones de estilos y patrones arquitectónicos existen rasgos comunes, algunos patrones coinciden con los estilos hasta en el nombre con que se les designa, por ejemplo, Arquitectura en capas, Tuberías y filtros y Modelo – Vista – Controlador. La diferencia fundamental entre ambos radica en el nivel de abstracción en que se aplican. Los estilos se encuentran en un plano de abstracción más alto, definiendo como configurar la arquitectura, mientras que los patrones están más cercanos al diseño, estos pueden representarse incluso mediante código en un lenguaje de programación determinado.

Carlos Reynoso y Nicolás Kiccillof en su trabajo: “Estilos y Patrones en la Estrategia de Arquitectura de Microsoft” [10] expresan: “En cuanto a los patrones de arquitectura, su relación con los estilos arquitectónicos es perceptible, pero indirecta y variable incluso dentro de la obra de un mismo autor”, sobre las diferencias entre ambos conceptos señalan:

- ✓ Existen claras convergencias entre ambos conceptos, aun cuando se reconoce que los patrones se refieren más bien a prácticas de re-utilización y los estilos conciernen a teorías sobre la estructuras de los sistemas a veces más formales que concretas.
- ✓ Quienes trabajan con estilos favorecen un tratamiento estructural que concierne más bien a la teoría, la investigación académica y la arquitectura en el nivel de abstracción más elevado, mientras que quienes trabajan con patrones se ocupan de cuestiones que están más cerca del diseño, la práctica, la implementación, el proceso, el refinamiento, el código.

A continuación se describen algunos estilos arquitectónicos empleados en diferentes sistemas de visualización médica:

1.2.1.1 Arquitecturas basadas en componentes

Un componente es una unidad de composición de aplicaciones, que posee un conjunto de interfaces especificadas contractualmente y dependencias del contexto explícitas, puede ser desplegado de forma independiente y está sujeto a la composición por terceras partes [11].

El estilo de arquitectura basada en componentes, describe un acercamiento al diseño de sistemas como un conjunto de componentes que exponen interfaces bien definidas y que colaboran entre sí para resolver el problema (ver Fig. 1).

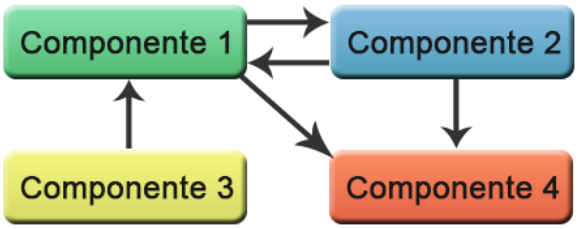

Fig. 1 Arquitectura basada en componentes.

Este estilo se utiliza para diseñar aplicaciones a partir de componentes individuales, enfatiza la descomposición del sistema en componentes lógicos o funcionales y define una aproximación al diseño a través de componentes que se comunican mediante interfaces que contienen métodos, eventos y propiedades [12].

El uso de este estilo facilita el despliegue, pues permite sustituir un componente por su nueva versión sin afectar a otros componentes o al sistema y favorece la reusabilidad de los componentes independientes del contexto, permitiendo que se empleen en otras aplicaciones y sistemas.

Entre los principales inconvenientes que puede traer el desarrollo basado en componentes se encuentran:

- Pueden afectar el desempeño del sistema al cual se integran.
- Si el sistema debe aceptar diferentes versiones de un mismo componente, es necesario realizar un proceso de gestión, para lo que se necesita código especializado para el manejo de la evolución de los componentes.
- Los servicios que ofrecen los componentes son usualmente definidos en tiempo de compilación, lo que puede ocasionar problemas de integración con el sistema en desarrollo, para modificar los componentes una vez creados es necesario manipular su implementación y posteriormente recompilarlos.

1.2.1.2 Arquitecturas en capas

Este estilo pertenece a la categoría Estilos de Llamada y Retorno y define una organización jerárquica tal que cada capa proporciona servicios a la capa inmediatamente superior y se sirve de las prestaciones que le brinda la inmediatamente inferior (ver Fig. 2) [10].

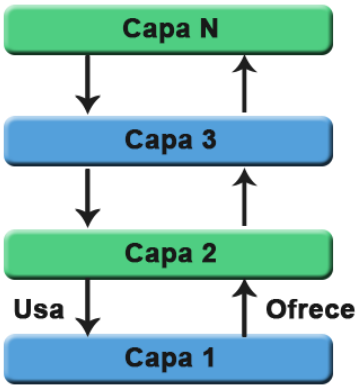

Fig. 2 Arquitectura en capas.

Entre las principales características de este estilo se encuentran la descomposición de los servicios de forma que la mayoría de interacciones ocurre solo entre capas vecinas. Los componentes de cada capa se comunican con los componentes de otras capas a través de interfaces conocidas y cada nivel agrega las responsabilidades y abstracciones del nivel inferior [12].

El uso de este estilo trae numerosos beneficios, por ejemplo:

- Abstracción: los cambios se realizan a alto nivel y se puede incrementar o reducir el nivel de abstracción que se usa en cada capa del modelo.
- Aislamiento: se pueden realizar actualizaciones en el interior de las capas sin que esto afecte el resto del sistema.
- Rendimiento: distribuyendo las capas en distintos niveles físicos se puede mejorar la escalabilidad, la tolerancia a fallos y el rendimiento.
- Facilidad de pruebas: cada capa tiene una interfaz definida sobre la que realizar las pruebas.

Algunas desventajas que representa la utilización de este estilo son [10]:

- Muchos problemas no se adaptan naturalmente a una estructura de capas.
- Los cambios en las capas de bajo nivel tienden a filtrarse hacia las de alto nivel.
- La arquitectura en capas ayuda a controlar y encapsular aplicaciones complejas, pero puede complicar las aplicaciones simples.

1.2.1.3 Tuberías y filtros

Este estilo pertenece a la clase Estilos de Flujo de Datos. Una tubería (*pipeline*) es un conjunto de elementos de procesamiento de datos conectados en serie, de modo que la salida de un elemento es la entrada del siguiente. Cada etapa del procesamiento se encapsula en un filtro. Los datos se transmiten a través de tubos entre filtros adyacentes [13]. La Fig. 3 muestra el *pipeline* típico de un software para la visualización tridimensional de imágenes médicas digitales.

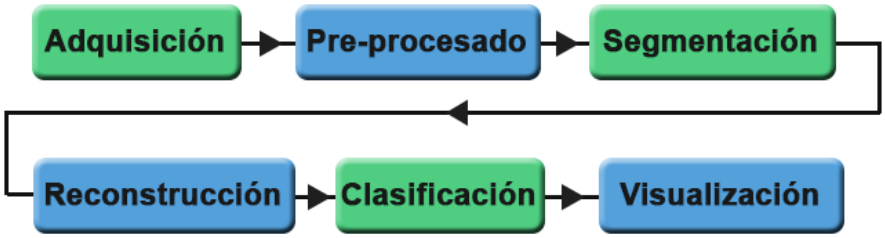

Fig. 3 Proceso de visualización empleando tuberías y filtros.

Entre las principales ventajas de este estilo se encuentran: la reutilización (los filtros no dependen unos de otros y se pueden utilizar en cualquier sistema donde se transmitan los mismos tipos de datos), el bajo acoplamiento, la concurrencia y facilita además estimar la velocidad de procesamiento del sistema.

Entre las principales desventajas del estilo se encuentran [10]:

- Presenta deficiencias con el manejo de construcciones condicionales, bucles y otras lógicas de control de flujo. Agregar un paso suplementario afecta el rendimiento de cada ejecución de la tubería, lo que puede influir negativamente en el desempeño del sistema.
- El estilo no es apto para manejar situaciones interactivas, sobre todo cuando se requieren actualizaciones incrementales de la representación en pantalla.

1.3 Patrones de diseño

Los patrones de diseño trabajan a una escala intermedia y son independientes del lenguaje de programación que se utilice. Su aplicación no tiene efectos en la estructura fundamental del sistema (arquitectura), pero puede tener una fuerte influencia sobre la arquitectura de un subsistema [14]. Definen un esquema de refinamiento de los subsistemas o componentes dentro de un sistema, o las relaciones entre estos.

Los patrones de diseño describen una estructura común y recurrente de componentes interrelacionados, que resuelve un problema general de diseño dentro de un contexto particular. Se encuentran divididos en tres categorías: creacionales, estructurales y de comportamiento. En la categoría de patrones creacionales se encuentran los que se relacionan con el proceso de creación de objetos, los estructurales tratan con la composición de clases u objetos y los de comportamiento caracterizan las formas en la que los objetos o clases interactúan y la distribución de responsabilidades entre estos [15]. En la Tabla 1 se muestra la distribución de los patrones de diseño en las tres categorías antes mencionadas, según lo que se observa en el libro *Design Patterns: Elements of Reusable Object – Oriented Software.*

Tabla 1 Distribución de patrones de diseño por categorías.

Patrones creacionales	Patrones estructurales	Patrones de comportamiento
Abstract Factory	*Adapter*	*Chain of Responsability*
Builder	*Bridge*	*Command*
Factory Method	*Composite*	*Interpreter*
Prototype	*Decorator*	*Iterator*
Singleton	*Facade*	*Mediator*
	Flyweight	*Memento*
	Proxy	*Observer*
		State
		Strategy
		Template Method
		Visitor

A continuación se describen algunos de los principales patrones de diseño.

1.3.1 Patrón de diseño *Adapter*

Convierte la interfaz de una clase en otra interfaz que el cliente puede interpretar. Permite el trabajo conjunto de clases con interfaces incompatibles [15].

Este patrón puede implementarse de dos formas diferentes:

1. Como adaptador de clase, empleando herencia múltiple para adaptar una interfaz a otra (ver Fig. 4).

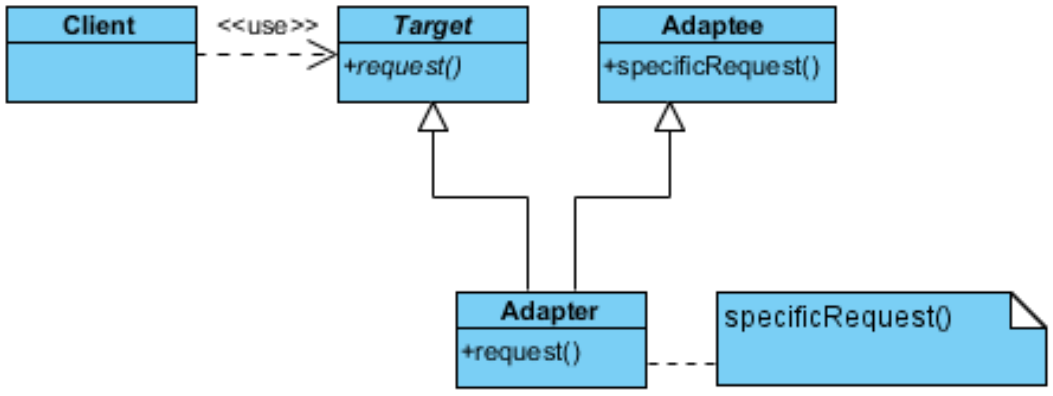

Fig. 4 Estructura del patrón de diseño *Adapter* como adaptador de clases.

2. Como adaptador de objetos, basado en la composición de los mismos (ver Fig. 5).

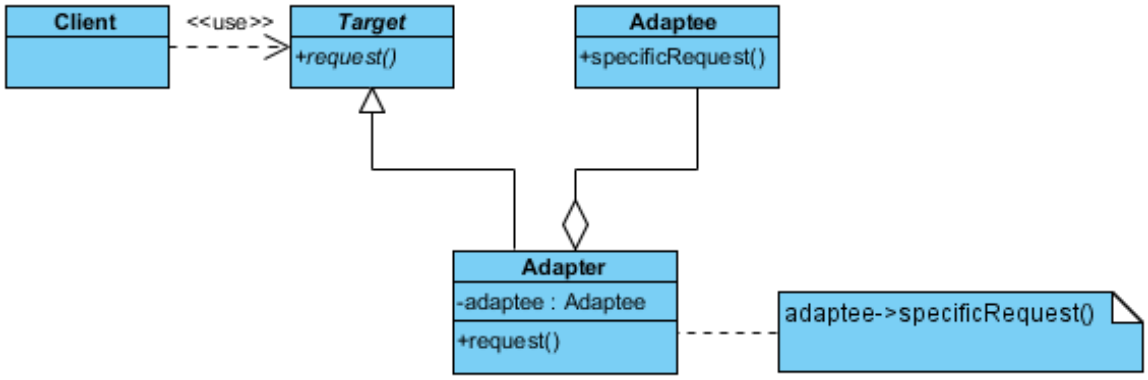

Fig. 5 Estructura del patrón de diseño *Adapter* como adaptador de objetos.

A continuación se describe el objetivo principal de cada una de estas clases:

Client**:* Clase que se comunica con la interfaz de destino (Target***).

Target**:* Define la interfaz específica de dominio que el Cliente (Client***) utiliza.

Adapter: Adapta la interfaz adaptada (***Adaptee***) a la interfaz de destino (***Target***).

Adaptee: Representa una interfaz existente que necesita adaptación para su comprensión por parte del cliente.

1.3.2 Patrón de diseño *Observer*

Define una relación de dependencia de uno a muchos entre objetos, cuando el objeto observado cambia su estado, todos sus objetos dependientes son notificados y actualizados automáticamente [15]. La estructura de este patrón se observa en la siguiente figura:

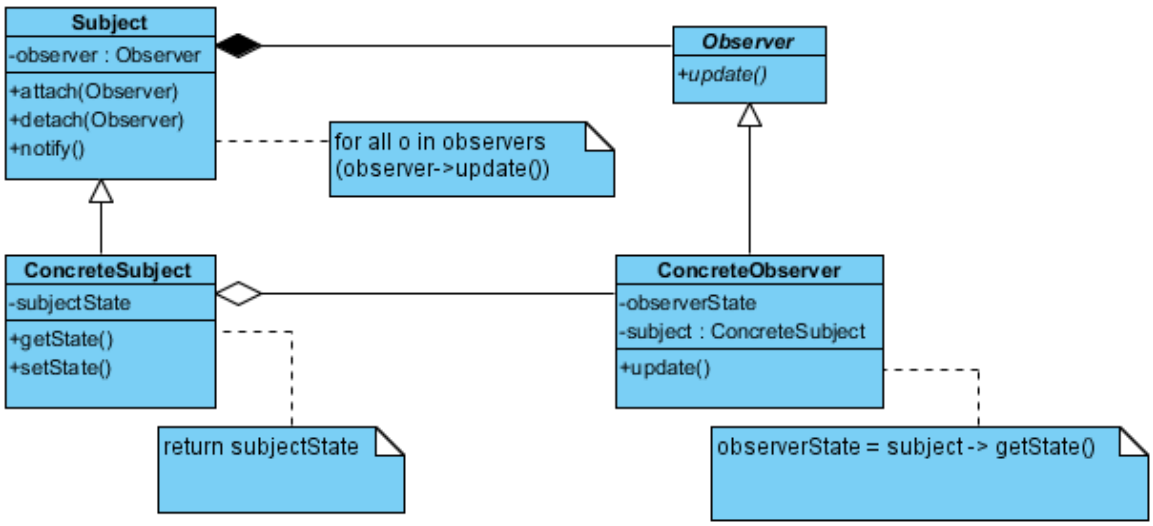

Fig. 6 Estructura del patrón de diseño *Observer*.

El objetivo de las clases que intervienen en este patrón es:

Subject: Tiene una lista variable de observadores, proporciona una interfaz para adicionar y eliminar los objetos observadores.

Observer: Define una interfaz de actualización para los objetos, estos deben ser notificados de los cambios del objeto observado (***Subject***).

ConcreteSubject: Almacena estados de interés para los observadores concretos (***ConcreteObserver***). Envía una notificación a sus observadores concretos cuando sucede un cambio de su estado.

ConcreteObserver: Mantiene una referencia a un objeto de tipo ***ConcreteSubject***. Guarda un estado que debe permanecer sincronizado con el estado del sujeto. Implementa la interfaz de actualización ***Observer*** para mantener su estado consistente con el del sujeto.

1.3.3 Patrón de diseño *Singleton*

Este patrón consiste en asegurar que una clase solo tenga una instancia y provee un punto de acceso global a esta [15]. En la Fig. 7 se observa la estructura de este patrón.

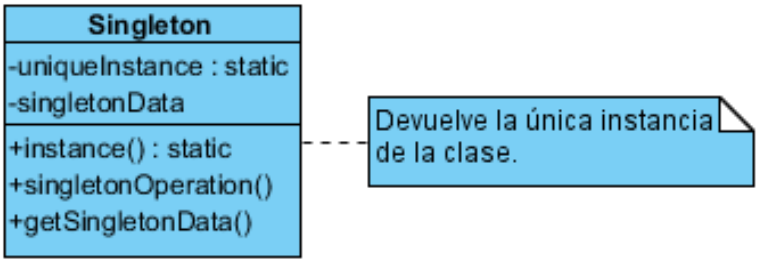

Fig. 7 Estructura del patrón de diseño *Singleton*.

La clase ***Singleton*** tiene como objetivo definir la operación ***instance()***, esta permite a los clientes crear y acceder a la única instancia de la clase.

1.3.4 Patrón de diseño *State*

Permite a un objeto alterar su comportamiento cuando su estado interno cambia, el resultado aparenta un cambio de clase del objeto [15]. En la Fig. 8 se observan las clases que intervienen en el patrón y la estructura del mismo.

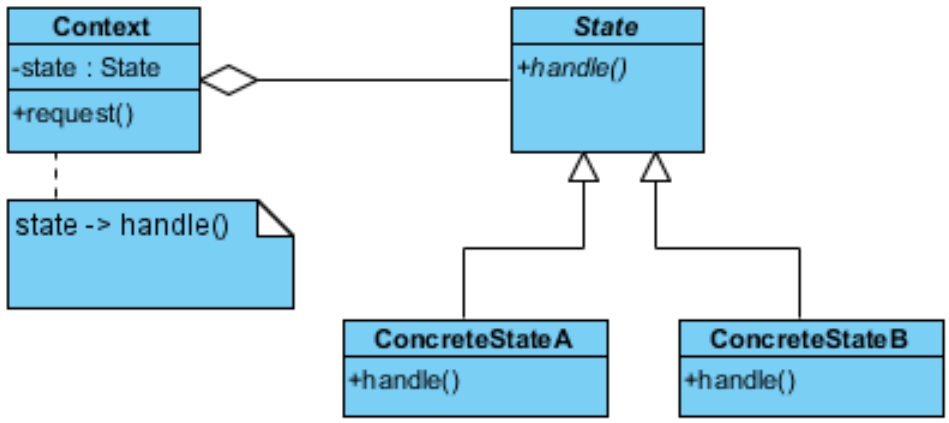

Fig. 8 Estructura del patrón de diseño *State*.

Las clases que intervienen en este patrón tienen como objetivos:

Context: Define una interfaz de interés para los clientes. Mantiene una instancia de un estado concreto que representa el estado actual.

State: Define una interfaz para encapsular el comportamiento asociado con estados particulares del contexto.

ConcreteState: Cada subclase implementa un comportamiento asociado con un estado particular del contexto.

1.3.5 Patrón de diseño *Factory Method*

Define una interfaz para crear un objeto, pero delega a las subclases la decisión del tipo de objeto a instanciar [15]. La estructura del patrón *Factory Method* se observa en la Fig. 9.

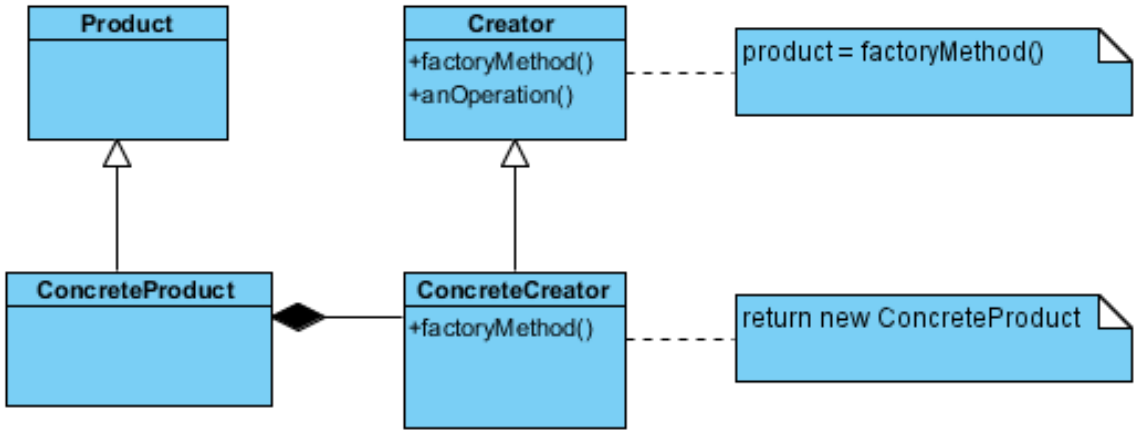

Fig. 9 Estructura del patrón de diseño *Factory Method*.

Los objetivos de las clases que conforman el patrón son:

Product: define la interfaz de los tipos de objetos que se pueden crear.

ConcreteProduct: implementa la interfaz ***Product*** y representa un producto concreto.

Creator: declara el método que se encarga de instanciar los productos (***factoryMethod()***), este retorna un objeto de tipo ***Product***.

ConcreteCreator: reimplementa el método encargado de instanciar los productos (***factoryMethod()***) y retorna una instancia de un producto concreto (***ConcreteProduct***).

1.4 Patrones de software para la asignación general de responsabilidades

Los patrones de software para la asignación general de responsabilidades (GRASP, por sus siglas en inglés[2]), describen los principios fundamentales del diseño de objetos y la asignación de responsabilidades entre objetos, el empleo de este tipo de patrones se considera una buena práctica de los principios de Programación Orientada a Objetos, estos son [16]:

- Experto en Información o Experto.
- Creador.
- Bajo Acoplamiento.
- Alta Cohesión.
- Controlador.
- Polimorfismo.
- Fabricación Pura.
- Indirección.
- Variaciones Protegidas.

A continuación se describen los patrones más propensos a emplearse según la situación problémica planteada para la investigación.

Experto:

Sigue el principio de asignar las responsabilidades a la clase que mayor información contenga. Su empleo trae como beneficios mantener el encapsulamiento de la información (los objetos utilizan su propia información para llevar a cabo las tareas), se distribuye el comportamiento entre las clases que contienen la información requerida (estimula las definiciones de clases cohesivas que son más fáciles de entender y mantener), bajo acoplamiento y alta cohesión [16].

Bajo Acoplamiento:

Sigue el principio de asignar una responsabilidad de manera que el acoplamiento permanezca bajo. El acoplamiento es una medida de la fuerza con que un elemento depende de otro, por tanto un elemento con bajo acoplamiento depende de la menor cantidad de elementos posibles, estos elementos pueden ser clases, subsistemas y sistemas, por ejemplo. Emplear el bajo acoplamiento posibilita que el cambio de un componente afecte en menor medida a los restantes y favorece la reutilización de los elementos [16].

Alta Cohesión:

Sigue el principio de asignar una responsabilidad de manera que la cohesión permanezca alta. La cohesión es una medida de la fuerza con la que se relacionan los elementos y del grado de

[2] GRASP: *General Responsibility Assignment Software Patterns.*

focalización de las responsabilidades de estos. Un elemento con responsabilidades altamente relacionadas, y que no realiza diversas funciones, tiene alta cohesión.

Como regla empírica, una clase con alta cohesión tiene un número relativamente pequeño de métodos, con funcionalidad altamente relacionada, y no realiza demasiadas funciones. Una mala cohesión causa, normalmente, un mal acoplamiento, y viceversa. El empleo de alta cohesión incrementa la claridad y facilita la comprensión del diseño, simplifica el mantenimiento y las mejoras, facilita el soporte para emplear bajo acoplamiento e incrementa la reutilización [16].

Polimorfismo:

El polimorfismo es un principio fundamental para diseñar cómo se organiza el sistema a la hora de gestionar variaciones similares. Cuando las alternativas o comportamientos relacionados varían según el tipo (clase), se asigna la responsabilidad para el comportamiento, utilizando operaciones polimórficas, al candidato adecuado, esto se obtiene normalmente implementando una interfaz común por todos los candidatos. Según el polimorfismo, un diseño basado en la asignación de responsabilidades puede extenderse para manejar nuevas variaciones. Sus principales ventajas consisten en añadir de forma simple las extensiones necesarias para nuevas variaciones e introducir las nuevas implementaciones sin afectar a los clientes [16].

1.5 Arquitectura de proyectos similares

Durante el desarrollo de la investigación se analizaron productos relacionados con el procesamiento de imágenes médicas digitales. Se seleccionaron tres productos que realizan, en cierta medida, funciones homólogas a las de Vismedic. Durante la selección se tuvo en cuenta que se desarrollaran con código abierto (*open source*), la frecuencia de sus publicaciones y la actividad de sus respectivas comunidades.

El estudio de los productos *Volume Rendering Engine (*Voreen*), Visualization Toolkit* (VTK) e *Insight Toolkit* (ITK) tuvo como objetivos fundamentales: determinar las características arquitectónicas que pueden ser reutilizables para la arquitectura de Vismedic y recopilar buenas prácticas de programación empleadas por proyectos establecidos en el campo. A continuación se detalla el análisis realizado a cada uno de los productos seleccionados.

1.5.1 VTK

VTK es un sistema de software ampliamente usado para el procesamiento y visualización de datos. Se emplea en cálculos científicos, análisis por imágenes médicas, geometría computacional y visualización de datos volumétricos, entre otros campos.

Como todo sistema de visualización, su objetivo fundamental es transformar los datos de entrada para hacerlos comprensibles por el sistema visor humano. Por tanto, uno de los requerimientos principales de VTK es la habilidad de crear tuberías de flujo de datos, capaces de adquirir, procesar y visualizar datos. VTK se diseñó como una herramienta altamente flexible, con

numerosos componentes intercambiables que pueden combinarse para procesar una amplia variedad de datos [17].

Arquitectura de tuberías y filtros

La arquitectura de VTK está compuesta por numerosos módulos o subsistemas [17], basados fundamentalmente en una arquitectura de tubería o flujo de datos. VTK posee tres clases de objetos fundamentales que intervienen en el proceso, estos son:

- ***vtkDataObject*:** Para la representación de datos.
- ***vtkAlgorithm*:** Encapsula algoritmos de procesamiento, transformación, filtrado o mapeo de objetos que representan datos.
- ***vtkExecutive*:** Empleado para ejecutar la red o tubería, posee un grafo de datos y algoritmos interrelacionados.

Aunque conceptualmente aparenta ser simple, la implementación de este tipo de arquitectura es considerada un reto. Una de las razones es que la representación de los datos puede ser arbitrariamente compleja, por ejemplo, algunos conjuntos de datos se componen de jerarquías o grupos de datos, por tanto la ejecución de la red requiere de un algoritmo de recursión o de búsqueda en estructuras jerárquicas.

Los objetos de tipo algoritmo introducen su propio nivel de complejidad. Algunos algoritmos necesitan múltiples entradas o salidas de diferentes tipos de datos, mientras que otros pueden operar localmente sobre los datos (ej., calcular el centro de una celda) o requerir información global, por ejemplo para calcular el histograma. En todos los casos los algoritmos deben recibir las entradas con el objetivo de producir sus salidas, esto se debe a que los datos deben estar disponibles para múltiples algoritmos, se recomienda que los algoritmos no interfieran en las entradas de otros algoritmos.

Finalmente, la ejecución de la red puede complicarse en dependencia de la estrategia empleada. En la mayoría de los casos se recomienda almacenar los resultados intermedios entre los algoritmos, esto disminuye el tiempo de cálculo cuando alguno de los elementos de la red cambia.

Subsistema de representación

A primera vista, VTK tiene un modelo de representación (*rendering*) orientado a objetos, relativamente simple, con clases correspondientes a los componentes clásicos de las escenas tridimensionales.

Como se observa en la Fig. 10, el proceso de visualización comienza con la entrada de los datos y culmina con la representación de una imagen en la pantalla. Una aplicación típica de visualización combina varios de los elementos antes mencionados, correspondientes a los diferentes elementos que aparecen en la pantalla. Estos pueden ser representaciones de los mismos datos iniciales, selecciones individuales de estos datos u objetos totalmente independientes.

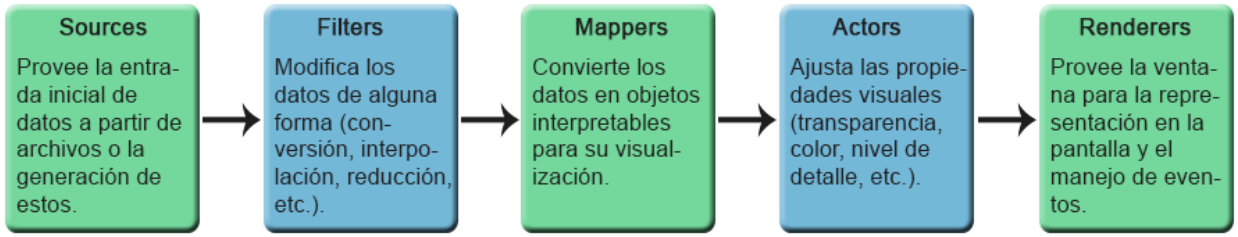

Fig. 10 *Pipeline* de visualización de VTK.

1.5.2 ITK

ITK es una biblioteca de clases para el análisis de imágenes que fue desarrollada por la iniciativa de la Biblioteca Nacional de Medicina de EE.UU. ITK puede pensarse como una enciclopedia utilizable de algoritmos para procesamiento de imágenes digitales (filtrado, segmentación y registro). La biblioteca fue desarrollada por un consorcio formado por universidades, empresas comerciales y numerosos colaboradores individuales de todo el mundo. El desarrollo de ITK inició en 1999 y, recientemente, después de su décimo aniversario, se sometió a un proceso de refactorización para eliminar el código redundante y darle forma para la próxima década [18].

En un problema típico de análisis de imágenes, el especialista tiene una imagen de entrada, debe mejorar algunas de las características de la imagen inicial, por ejemplo mediante reducción de ruido o aumento de contraste, y luego proceder a identificar elementos significativos en la imagen, tales como bordes y fronteras. Este tipo de procesamiento es ideal para una arquitectura de flujo de datos (*data pipeline*), como se muestra en la Fig. 11.

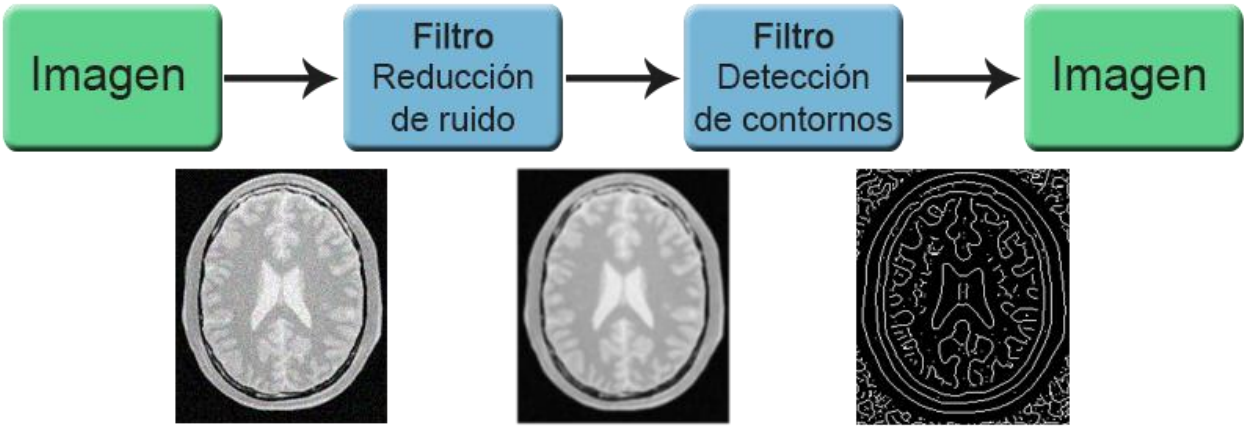

Fig. 11 *Pipeline* de procesamiento de imágenes.

Una de las características fundamentales de la arquitectura de ITK es la modularidad, esta característica posibilita la incorporación a la biblioteca de nuevos módulos desarrollados por la comunidad de procesamiento de imágenes. La mayoría de los problemas de análisis de imágenes reciben una o varias imágenes como entrada y aplican una combinación de filtros que mejoran o extraen información particular de una parte de la imagen. Por tanto no existe un único objeto de

procesamiento continuo, en lugar de esto se emplea la combinación de muchos objetos pequeños de procesamiento. La naturaleza estructural del problema implica la implementación de una larga colección de filtros de procesamiento que pueden ser combinados de diferentes formas.

La arquitectura modular de ITK posibilita o facilita:

- Reducción y claridad de las dependencias cruzadas.
- Adopción de código contribuido por la comunidad.
- Evaluación de métricas de calidad por módulo.
- Construcción y despliegue de componentes independientes de la biblioteca.
- Crecimiento continuo y adición progresiva de módulos.

El carácter secuencial de la mayoría de las tareas de análisis de imágenes, conduce de forma natural a la selección de una arquitectura de flujo de datos como espina dorsal para el procesamiento. La arquitectura de flujo de datos en ITK posibilita:

- **Concatenación de filtros:** concatena una serie de filtros para conformar una cadena de procesamiento que aplica las operaciones a las imágenes de entrada de forma secuencial.
- **Exploración de parámetros:** una vez conformada la cadena de procesamiento, se torna simple cambiar el valor de cualquier parámetro y analizar los efectos que dicho cambio ocasiona en la imagen final.
- **Flujo de memoria (*memory streaming*):** permite procesar imágenes grandes empleando solo sub-bloques, de esta forma se posibilita el procesamiento de las imágenes que exceden la capacidad de la memoria principal.

ITK contiene dos tipos de objetos principales para sostener la estructura básica de la arquitectura de flujo de datos, ellos son ***DataObject*** y ***ProcessObject***. ***DataObject*** es la abstracción de los objetos que contienen datos, por ejemplo imágenes o mallas geométricas. ***ProcessObject*** provee una abstracción para los filtros de imágenes que procesan dichos datos. Un mismo ***DataObject*** puede ser la entrada de diversos ***ProcessObject***.

Los objetos de tipo ***DataObject*** y ***ProcessObject*** se conectan entre sí como un efecto secundario de la conexión de la tubería. Desde el punto de vista del desarrollador, la tubería se conecta mediante una secuencia de llamadas a métodos pertenecientes a ***ProcessObject***, como se muestra en el siguiente fragmento de código.

```
median->SetInput(reader->GetOutput());
invert->SetInput(median->GetOutput());
gradient->SetInput(invert->GetOutput());
```

El diseño e implementación original de la arquitectura de flujo de datos de ITK se deriva de la arquitectura de VTK [18], la Fig. 12 muestra la jerarquía básica de clases que soporta la implementación de dicha arquitectura.

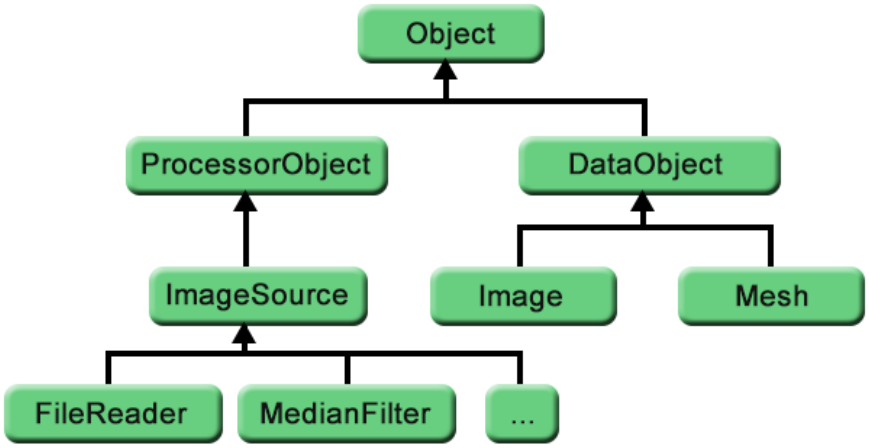

Fig. 12 Jerarquía básica de clases: *ProcessObject* y *DataObject*.

Especial atención debe prestarse a la relación entre las clases ***Object***, ***DataObject*** y ***ProcessObject***. En esta abstracción, cualquier objeto que se espera sea la entrada o salida de un filtro, debe heredar de la clase ***DataObject***. Todos los filtros que producen o consumen datos deben derivarse de ***ProcessObject***.

Una de las principales cualidades que se identificaron durante el análisis de ITK es su reusabilidad, esto se logra en la biblioteca empleando extensivamente los principios de programación orientada a objetos, en particular la creación de jerarquías de clases donde las funcionalidades comunes se agrupan en clases base. La adopción de programación genérica, empleando las plantillas de C++ (*templates*), le permite a ITK extender los comportamientos de las clases sin realizar cambios en la arquitectura base. El uso reiterado de macros escritas en C++ posibilita reusar fragmentos de código estándares que se necesitan en numerosos lugares de la biblioteca.

1.5.3 Voreen

Voreen es una biblioteca de código abierto para visualización de volúmenes. Posee alta flexibilidad a la hora de incorporar nuevas técnicas de visualización. Está implementada como una biblioteca multiplataforma (Windows, Linux y Mac.), usando OpenGL como biblioteca gráfica y GLSL como lenguaje de programación para los algoritmos basados en GPU (*Graphics Proccesing Unit*). Voreen se distribuye según los términos de la Licencia Pública General o GPL (acrónimo del inglés *General Public License*) [19].

El diseño de la biblioteca Voreen gira alrededor del concepto de redes de flujo de datos. Estas redes consisten en unidades modulares, llamadas procesadores (*processors*), que encapsulan los algoritmos de procesamiento y visualización de datos. Un procesador funciona con los datos de entrada que recibe a través de sus puertos de entrada (*inports*) y expone los resultados mediante sus puertos de salida (*outports*). El flujo de datos se establece mediante conexiones unidireccionales de *outports* a *inports*. Cada puerto transmite un cierto tipo de datos, como una

imagen 2D, un volumen 3D, datos geométricos, o una colección de objetos de estos tipos básicos. Los procesadores tienen además propiedades para la parametrización de los algoritmos encapsulados. Voreen ofrece varios tipos de propiedades que van desde tipos numéricos primitivos hasta parámetros complejos para la representación, como la posición de la cámara y la función de transferencia. Las propiedades pueden ser conectadas dentro y a través de procesadores, con el fin de sincronizar sus valores.

La principal fortaleza de la arquitectura de flujo de datos consiste en la flexibilidad a la hora de combinar los componentes. El comportamiento de un procesador está determinado únicamente por sus datos de entrada y la configuración de sus propiedades, por tanto, los procesadores pueden ser arbitrariamente combinados en redes con la única restricción de que los puertos conectados tienen que ser de igual tipo. Esto permite a los desarrolladores centrarse completamente en la implementación de nuevas técnicas y emplear los procesadores estándares para el procesamiento común de datos, representación e interacción con el usuario.

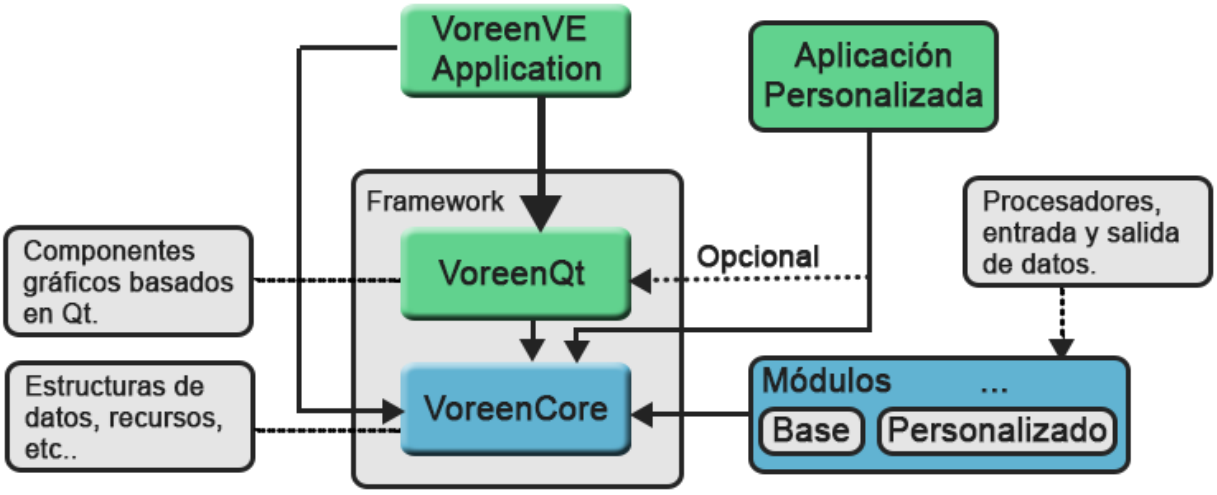

Fig. 13 Arquitectura de Voreen.

Como se observa en la Fig. 13, Voreen posee una arquitectura multi-capa que separa limpiamente el sistema de visualización de la capa de interfaz gráfica de usuario. La biblioteca ***VoreenCore*** contiene las estructuras de datos fundamentales y se encarga además de la administración de recursos (ej. imágenes o volúmenes de datos). Esta capa no posee dependencias para interfaces gráficas de usuario y está diseñada para emplearse directamente en aplicaciones externas. La biblioteca ***VoreenQt***, basada en el *framework* Qt, provee los componentes de interfaz gráfica de usuario, fundamentalmente los elementos gráficos asociados a los procesadores y sus propiedades. Esta capa puede emplearse también, pero de forma opcional, en aplicaciones externas. Por último, la aplicación ***VoreenVE***, provee el entorno de ejecución para los procesadores y la edición dinámica de la red de flujo de datos, lo que permite realizar, de forma visual, prototipos rápidos de nuevas aplicaciones.

Todos los componentes activos de Voreen están organizados en módulos, estos extienden las funcionalidades de la capa ***VoreenCore*** e incluyen específicamente:

- ***Processors*****:** unidades funcionales fundamentales que encapsulan los algoritmos de procesamiento y representación.
- ***Properties*****:** representan los parámetros empleados por los algoritmos de procesamiento y representación.
- ***Ports*****:** se emplean para conectar las entradas y salidas de los procesadores, puede ser del tipo enumerativo ***PortDirection***, que incluye ***inport*** y ***outport*** para los puertos de entrada y salida respectivamente.
- ***Data readers*** y ***writers*****:** encargados de leer y escribir los datos desde y hacia ficheros.

Los nuevos módulos que se incorporen a Voreen necesitan integrarse al *framework* en tiempo de compilación y registrarse en tiempo de ejecución. La integración en tiempo de compilación se realiza mediante un archivo de CMake, el nombre del archivo respeta la siguiente estructura: <nombredelmodulo>.cmake. En tiempo de ejecución, los recursos del módulo se registran mediante una clase que hereda de ***VoreenModule***.

Una de las principales novedades de Voreen es su editor de redes de flujo de datos, esta permite crear y modificar de forma visual la red de procesamiento, así como establecer las conexiones entre los puertos pertenecientes a los procesadores. La red permite observar gráficamente todo el procesamiento hasta la obtención de la representación final. En la Fig. 14 se observa una red de procesamiento que muestra un corte sagital a partir de un volumen de datos.

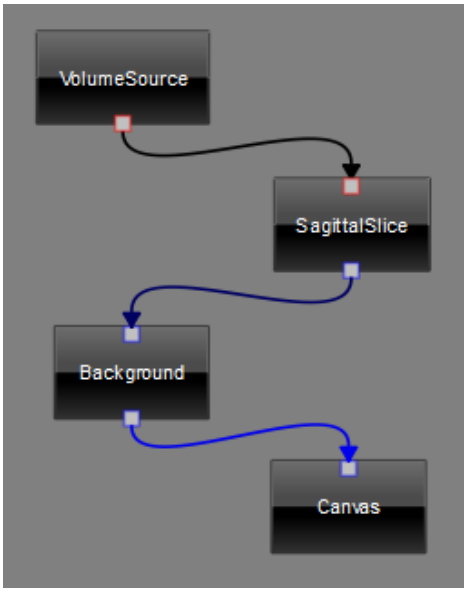

Fig. 14 Red de procesamiento de Voreen.

Comparado con VTK e ITK, Voreen representa un escalón superior en cuanto a extensibilidad y reusabilidad, además, la posibilidad de crear y modificar las redes de procesamiento en tiempo

de ejecución, aumenta la variedad de aplicaciones a desarrollar, incluso por usuarios finales que no intervienen en el proceso de desarrollo de software. Es necesario mencionar que Voreen no no presenta un uso bien definido de *plugins* con el objetivo de extender su funcionamiento, por tanto, las modificaciones que se realicen deben ser en tiempo de compilación, lo que implica que, una vez desplegada la biblioteca, necesita recompilarse para incorporar las nuevas funcionalidades.

1.6 Especificación OSGi para el desarrollo basado en componentes

OSGi, acrónimo del inglés *Open Services Gateway Initiative,* fue creado en marzo de 1999, como un sistema (o *framework*) modular para Java que especifica la forma de crear módulos y la manera en que estos interactuarán en tiempo de ejecución. Este *framework* proporciona a los desarrolladores un entorno orientado a servicios y basado en componentes, ofreciendo estándares para manejar el ciclo de vida completo del software [20]. La Fig. 15 muestra la arquitectura multi-capa presente en OSGi y la interrelación entre las mismas.

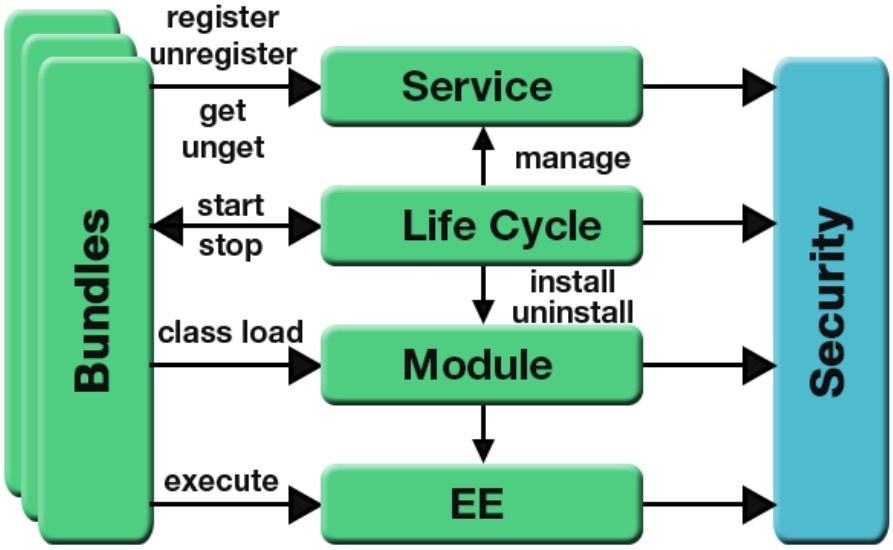

Fig. 15 Arquitectura multi-capa de OSGi.

La capa de seguridad (***Security Layer***) puede emplearse de manera opcional y está basada en la arquitectura de seguridad de Java 2. Provee una infraestructura para desplegar y manejar aplicaciones que tienen que ejecutarse en un entorno seguro y controlado. El empleo de esta capa contribuye a mejorar la integridad y la autentificación de los componentes de software, propiedades que permiten mantener los datos libres de modificaciones no autorizadas e identificar el generador de la información, respectivamente.

Un módulo es una aplicación empaquetada en un fichero JAR, que se despliega en una plataforma OSGi. Cada módulo contiene un fichero de metadatos organizado por pares de tipo [clave: valor], donde se describen las relaciones del módulo, por ejemplo los paquetes que

importa o exporta. Al desplegar un módulo dentro de una plataforma OSGi comienza su ciclo de vida, controlado por el propio *framework*.

La plataforma en sí misma, proporciona una capacidad limitada para crear aplicaciones modulares. El *framework* OSGi proporciona una solución genérica y estandarizada para la modularización de aplicaciones Java. ***Module Layer*** define la anatomía de los módulos, las reglas de interoperación entre los mismos y la arquitectura de carga de clases [20]. Entre las principales características que identifican esta capa de OSGi se encuentran:

- La unidad de despliegue de aplicación es el módulo, normalmente empaquetado en un archivo con extensión .jar.
- Soporte para distintas versiones de un mismo módulo, permite importar paquetes teniendo en cuenta su versión.
- Código declarativo específico para gestionar las dependencias (importación y exportación de paquetes).

La capa ***Life Cycle Layer*** se encarga de gestionar el ciclo de vida de un módulo dentro del *framework*. El ciclo de vida de un módulo se especifica mediante los estados por los que este puede pasar:

- ***Installed***: El módulo ha sido instalado correctamente.
- ***Resolved***: El módulo está listo para iniciar o ha sido detenido.
- ***Starting***: El módulo ha iniciado (se quedará en este estado hasta que sea necesario su activación).
- ***Active***: El módulo se ha activado correctamente y se encuentra en funcionamiento.
- ***Stopping***: El módulo ha sido detenido.
- ***Uninstalled***: El módulo ha sido desinstalado. No se puede cambiar a otro estado.

La capa de servicios (***Service Layer***) proporciona un modelo de programación dinámico para los desarrolladores de módulos, simplificando el desarrollo y despliegue de estos a través del desacople entre la especificación del servicio y su implementación. En OSGi, un servicio es un objeto Java que se registra bajo una o varias interfaces. Los módulos pueden registrar servicios, buscarlos o recibir notificaciones cuando el estado de un servicio ha sufrido alguna variación.

1.7 Análisis de la arquitectura actual

El proyecto Vismedic cuenta actualmente con dos productos fundamentales (Visualizador 2D y Visualizador 3D), derivados de un período de tres años de investigación y desarrollo. Durante este proceso, los cambios frecuentes de los requisitos funcionales y los atributos de calidad de los productos a desarrollar, condujeron a realizar numerosas modificaciones a la arquitectura de software del proyecto.

La arquitectura sobre la que se desarrollan actualmente los productos del proyecto se basa

fundamentalmente en el empleo de capas y módulos, con el objetivo de agrupar las funcionalidades y delimitar las responsabilidades de los elementos que componen las aplicaciones. Como se observa en la Fig. 16, la arquitectura cuenta con una capa que representa el núcleo básico (***Core***) para los productos a desarrollar, esta capa se encuentra dividida en módulos que ofrecen funcionalidades comunes para las aplicaciones de visualización de imágenes médicas digitales. Dentro del módulo ***Plugins***, perteneciente a la capa ***Core***, se definen las interfaces a implementar para extender las aplicaciones mediante *plugins*, es necesario señalar que, en estos momentos, solo el Visualizador 2D hace uso de estos.

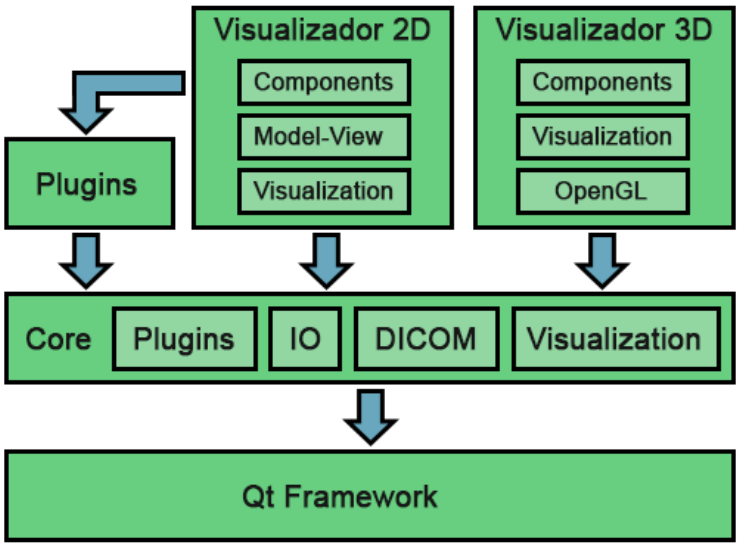

Fig. 16 Arquitectura actual del proyecto Vismedic.

Las aplicaciones concretas contienen módulos específicos que difieren en dependencia de los requisitos funcionales de cada una. Tanto el núcleo, como los Visualizadores 2D y 3D, contienen un módulo denominado ***Visualization***. El que se encuentra en el núcleo, define las interfaces para la incorporación de algoritmos de visualización tridimensional, por lo tanto, se emplea directamente por el Visualizador 3D; mientras que el módulo ***Visualization***, ubicado en el Visualizador 2D, implementa sus funcionalidades de forma independiente.

Luego del análisis realizado, se comprobó que la arquitectura actual presenta un fuerte acople y alto nivel de dependencia entre los componentes y módulos que la integran, esto se evidenció a la hora de adicionar nuevas funcionalidades o modificar las existentes, por ejemplo: para adicionar un nuevo algoritmo de visualización tridimensional, se necesita implementar la interfaz ***RenderAlgorithm*** (ubicada en el módulo ***Visualization*** de la capa ***Core***), registrar explícitamente el nuevo algoritmo en el contexto de visualización de OpenGL y ubicar en la barra de herramientas principal, en caso de ser necesario, un enlace a la ventana de configuración del nuevo algoritmo. Para realizar el despliegue con el nuevo algoritmo adicionado es necesario recompilar el producto y posteriormente reinstalarlo en cada una de las estaciones de trabajo. La arquitectura actual

afecta además, en cierta medida, los siguientes atributos de calidad:

Portabilidad: la biblioteca de clases DCMTK presenta problemas de portabilidad para el sistema operativo Windows, lo que provocó que la primera versión de los visualizadores estuviera disponible solo para Linux.

Modificabilidad – Mantenibilidad: la actualización de los componentes conlleva, como mínimo, a repetir el proceso completo de despliegue.

Reusabilidad – Escalabilidad: Las interfaces definidas en el módulo ***Plugins***, perteneciente a la capa ***Core***, solo ofrece soporte para algoritmos de filtrado de imágenes bidimensionales, lo que limita su uso para productos con características tridimensionales como el Visualizador 3D.

1.8 Consideraciones parciales

El estudio realizado permitió comprender la importancia de la arquitectura de software para el desarrollo de sistemas informáticos. Se analizaron las principales ventajas de la utilización de estilos y patrones arquitectónicos como base para el diseño e implementación de una arquitectura de software. Se caracterizaron diferentes patrones de diseño y de asignación general de responsabilidades.

El estudio de los productos Voreen, VTK e ITK permitió determinar las características arquitectónicas que pueden ser reutilizables en la arquitectura de Vismedic. Por otra parte, el análisis de la especificación OSGi posibilitó definir los elementos de las interfaces a implementar por los componentes independientes del sistema y las relaciones de estos en tiempo de ejecución.

CAPÍTULO 2. SOLUCIÓN PROPUESTA

En este capítulo se describe la arquitectura propuesta a partir del modelo “4+1” Vistas de la Arquitectura de Software. Se especifica la metodología de software y las herramientas que se utilizarán durante el diseño e implementación de la arquitectura propuesta y se presentan los requisitos funcionales y las restricciones arquitectónicas que se deben satisfacer.

2.1 Metodologías y herramientas de desarrollo

Para el desarrollo de la arquitectura propuesta para el proyecto Vismedic se utilizó como metodología de desarrollo de software RUP (acrónimo el inglés *Rational Unified Process*) con el lenguaje de modelado UML (acrónimo del inglés *Unified Modeling Language*), la herramienta CASE [3] *Visual Paradigm for UML* 8.0 y el *framework* de desarrollo QT 5.0 utilizando como lenguaje de programación C++. A continuación se detallan los elementos más significativos de estos.

2.1.1 Metodología de desarrollo de software RUP

Se utiliza como metodología de desarrollo de software RUP. Sus características fundamentales se basan en un desarrollo centrado en la arquitectura, iterativo e incremental y guiado por casos de uso. RUP propone una guía para establecer la línea base de la arquitectura de un proyecto de desarrollo de software basada en la modelación de las 4 + 1 vistas de la arquitectura. Proporciona una forma disciplinada para la asignación de tareas y responsabilidades en una organización de desarrollo de software, su propósito es asegurar la producción de software de alta calidad que se ajuste a las necesidades de los usuarios finales.

2.1.2 Lenguaje de modelado UML

La metodología RUP utiliza como lenguaje de modelado UML, este lenguaje está compuesto por diversos elementos gráficos que se combinan para conformar diagramas y tiene como principal objetivo especificar, construir, documentar y visualizar los artefactos que se crean durante el desarrollo de un sistema de software.

2.1.3 *Visual Paradigm for UML* 8.0

Como herramienta de modelado se utilizó *Visual Paradigm for UML* 8.0. Esta herramienta CASE posee soporte multiplataforma así como licencia comercial y gratuita, puede ser utilizada durante todo el ciclo de vida del desarrollo de software para modelar los diferentes diagramas y artefactos que se generan durante el desarrollo de un producto, posee capacidades de ingeniería directa e inversa, permitiendo convertir código fuente de programas, archivos ejecutables y binarios en

[3] CASE: *Computer Aided Software Engineering,* en español, Ingeniería de Software Asistida por Computadora

modelos UML, entre otras características.

2.1.4 *Framework* de desarrollo Qt 5.0

Qt es un *framework* multiplataforma para el desarrollo de aplicaciones. Utiliza C++ de manera nativa, pero ofrece soporte para otros lenguajes como Python, Java, C#, Ruby, entre otros. Ofrece una suite de aplicaciones para facilitar y agilizar las tareas de desarrollo, estas son:

- Qt Assistant: visualiza la documentación oficial de Qt.
- Qt Designer: herramienta para crear interfaces de usuario.
- Qt Linguist: herramienta para la traducción de aplicaciones.
- Qt Creator: IDE para el lenguaje C++, especialmente diseñado para Qt, integra las primeras dos herramientas mencionadas.

2.1.5 Lenguaje de programación C++

Como lenguaje de programación se utilizó C++, lenguaje por excelencia para las aplicaciones de realidad virtual que hace uso eficiente del paradigma de Programación Orientada a Objetos. Permite un excelente control de la memoria y una buena administración de los recursos de la computadora. Dentro de las principales ventajas que presenta se encuentran:

- **Difusión**: al ser uno de los lenguajes más empleados en la actualidad, posee gran número de usuarios y tiene una excelente bibliografía.
- **Versatilidad**: C++ es un lenguaje de propósito general, se puede emplear para resolver cualquier tipo de problema.
- **Portabilidad**: se encuentra estandarizado, por tanto, el mismo código fuente puede ser compilado en diferentes plataformas.
- **Eficiencia**: C++ es ampliamente conocido como uno de los lenguajes más eficientes en tiempo de ejecución.
- **Herramientas**: existen numerosos compiladores, depuradores y bibliotecas de clases basadas en este lenguaje.

2.2 Modelo del dominio

El modelo del dominio es una representación visual de los conceptos u objetos del mundo real significativos en el entorno del sistema. Se realiza con el objetivo de comprender el contexto del sistema y por tanto los requisitos del sistema que se desprenden de dicho contexto [21].

El entorno en que está enmarcado el problema, se puede describir cuando el **usuario** interactúa con el **entorno de ejecución**, a través del cual puede activar o desactivar los ***plugins*** existentes. El **entorno de ejecución** proporciona el ambiente donde los ***plugins*** pueden ejecutarse. Cada ***plugin*** contiene un objeto de tipo **procesador**, que se encarga de procesar las **imágenes** médicas en formato DICOM, o un **visualizador** que muestra la **imagen** resultante.

En la Fig. 17 se muestra mediante un modelo del dominio, las relaciones entre los elementos del entorno descrito anteriormente, con el objetivo de facilitar la comprensión de sus principales conceptos.

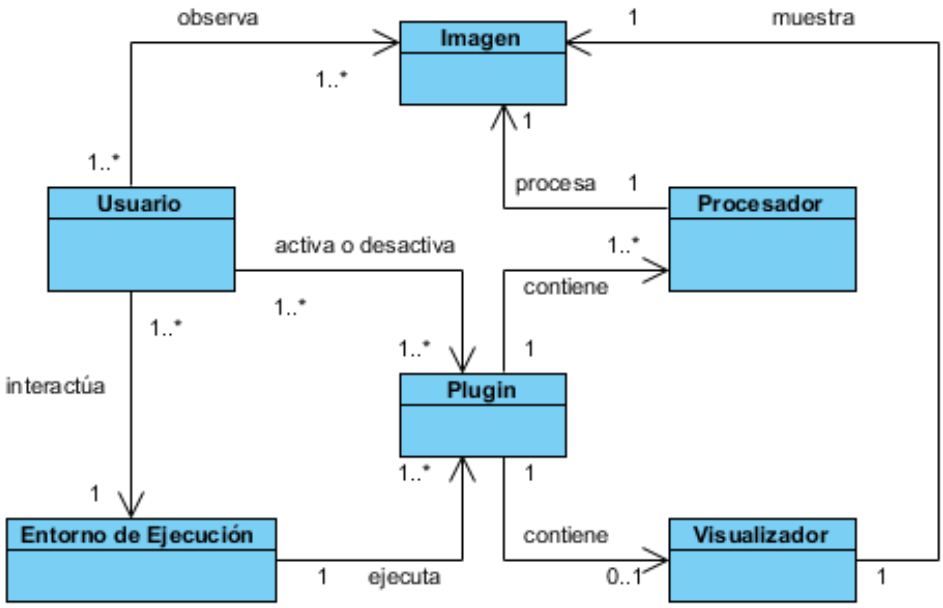

Fig. 17 Modelo del dominio.

Para la mejor comprensión del modelo a continuación se explican cada uno de los conceptos utilizados:

Usuario: Persona que interactuará con la aplicación que se desarrolle.

Entorno de Ejecución: Entorno que permite la incorporación de *plugins* y la personalización de los productos.

***Plugin*:** Módulo de software que añade una característica o un servicio específico al sistema.

Procesador: Encapsula algoritmos de procesamiento, comprende algoritmos para la lectura, filtrado, segmentación y visualización de imágenes.

Visualizador: Se encarga de mostrar en pantalla la imagen o la representación volumétrica.

Imagen: Las imágenes médicas digitales en formato DICOM constituyen los tipos de imágenes más utilizados en el proyecto Vismedic. Estas se obtienen como resultado de estudios orientados a los pacientes a través de Tomografías Axiales Computarizadas (TAC) o Resonancias Magnéticas (RM).

2.3 Requisitos funcionales

Un requerimiento es una condición o capacidad que el sistema debe cumplir para satisfacer un contrato, norma, especificación u otro documento formal, facilitando el entendimiento entre clientes y desarrolladores. Los requisitos funcionales especifican acciones que debe poder

realizar un sistema, sin tener en cuenta las restricciones físicas y se mantienen invariables sin importar con que propiedades o cualidades se relacionen [21]. A continuación se presentan los requisitos funcionales que intervienen en el desarrollo de la arquitectura de software propuesta y una breve descripción de los mismos (ver Tabla 2):

Tabla 2 Requisitos funcionales.

No	Nombre	Descripción
RF. 1	Adicionar *plugin.*	El sistema debe permitir al usuario seleccionar de un directorio el *plugin* que desee añadir a la aplicación.
RF. 2	Deshabilitar *plugin.*	El sistema debe mostrar al usuario el listado de los *plugins* activos en la aplicación para que seleccione el que desea deshabilitar.
RF. 3	Habilitar *plugin.*	El sistema debe mostrar al usuario el listado de los *plugins* que están deshabilitados para que seleccione el que desea habilitar.
RF. 4	Eliminar *plugin.*	El sistema debe mostrar el listado de los *plugins* presentes en la aplicación (activos o no) para permitir al usuario eliminar el *plugin* que desee.
RF. 5	Adicionar procesador.	El sistema debe permitir al usuario adicionar un procesador a la escena de procesamiento.
RF. 6	Modificar procesador.	El sistema debe permitir al usuario modificar los parámetros de configuración de un procesador en caso que este lo requiera.
RF. 7	Eliminar procesador.	El sistema debe permitir al usuario eliminar un procesador adicionado a la escena de procesamiento.
RF. 8	Concatenar procesadores.	El sistema debe permitir al usuario concatenar varios procesadores.
RF. 9	Ejecutar red de procesamiento.	El sistema debe brindar la opción al usuario de ejecutar la red de procesamiento que se encuentre en la escena (la red puede estar formada por uno o varios procesadores).

2.4 Restricciones arquitectónicas

La definición de una arquitectura de software debe poseer un conjunto de reglas para el desarrollo de un software, en este contexto, las reglas se denominan restricciones arquitectónicas. A

continuación se relacionan las restricciones arquitectónicas que debe cumplir la propuesta de solución:

- La arquitectura debe garantizar que los productos que se desarrollen sean multiplataforma (Linux (Ubuntu 13.04) y Windows 7 o superior).
- La arquitectura debe permitir la actualización o incorporación de compontes de forma natural.
- Se debe utilizar como lenguaje de programación C++ y como *framework* de desarrollo Qt.
- Las prestaciones de hardware dependerán de los requerimientos no funcionales de los productos que se desarrollen utilizando la arquitectura propuesta.

2.5 Descripción general de la arquitectura

A partir del estudio realizado se propone, para el proyecto Vismedic, una arquitectura basada en la combinación de los estilos arquitectónicos: arquitectura basada en capas, arquitectura basada en componentes y que haga uso de una red de flujo de datos (estilo arquitectónico tuberías y filtros) para el procesamiento y visualización de las imágenes. En la Fig. 18 se observa la distribución de las capas presentes en la arquitectura propuesta.

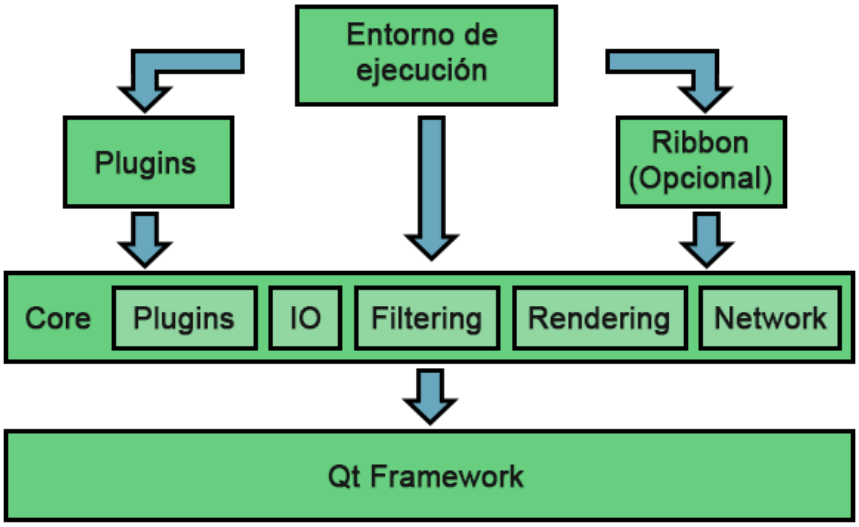

Fig. 18 Distribución de las capas en la arquitectura propuesta.

La capa ***Qt Framework*** ofrece el soporte para el manejo de plugins y para la interfaz gráfica de usuario. En la capa ***Core*** se encuentran ubicados los módulos y componentes comunes a todas las aplicaciones a desarrollar. Esta capa se encuentra dividida por módulos, dentro de los cuales se pueden mencionar: ***Plugins*** (contiene las interfaces a implementar por los *plugins* concretos que se deseen adicionar) e ***IO*** (para entrada y salida de datos). La capa ***Ribbon*** permite emplear de forma opcional la biblioteca *ribbon* desarrollada en el proyecto. Esta biblioteca posibilita crear interfaces gráficas de usuario empleando una cinta de opciones similar a los productos de

Microsoft Office 2007 – 2013. La capa ***Entorno de Ejecución*** provee el entorno donde se ejecutarán los *plugins*, así como las funcionalidades de activación y desactivación de estos. Por último la capa ***Plugins*** permite la creación de elementos de extensión para las aplicaciones, el control de su ciclo de vida dentro del entorno de ejecución, así como su versionado y resolución de dependencias internas, las que se gestionarán teniendo en cuenta lo propuesto por la especificación OSGi (ver sección 1.6).

2.6 Vistas arquitectónicas.

Para la descripción de la arquitectura de software propuesta se utilizará el modelo "4+1" Vistas de la Arquitectura de Software, propuesto por Kruchten [4]. En este modelo se define una vista arquitectónica como una descripción simplificada o abstracción de un sistema, desde una perspectiva específica, que cubre intereses particulares y omite entidades no relevantes a esa perspectiva. Las vistas que componen este modelo se observan en la Fig. 19.

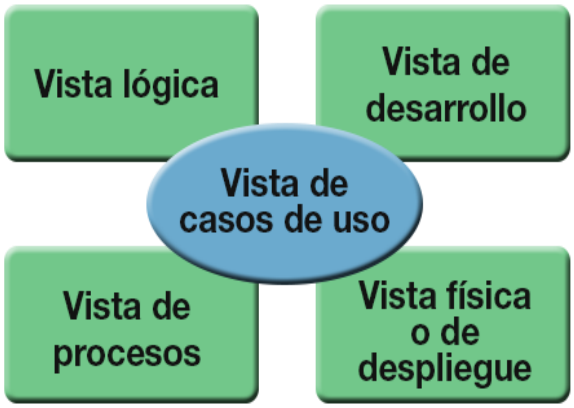

Fig. 19 El modelo de 4+1 vistas de la arquitectura de software.

2.6.1 Vista de casos de uso.

La vista de casos de uso en el marco arquitectónico, representa los casos de uso arquitectónicamente significativos (CUAS), estos son los que representan elementos críticos o funcionalidades imprescindibles para el sistema. Su representación es igual a la de un modelo de casos de uso, donde solo se representan los CUAS. Un modelo de casos de uso es un modelo del sistema que contiene actores, casos de uso y sus relaciones [21]. La Fig. 20 ilustra la vista de casos de uso correspondiente a la arquitectura de software propuesta.

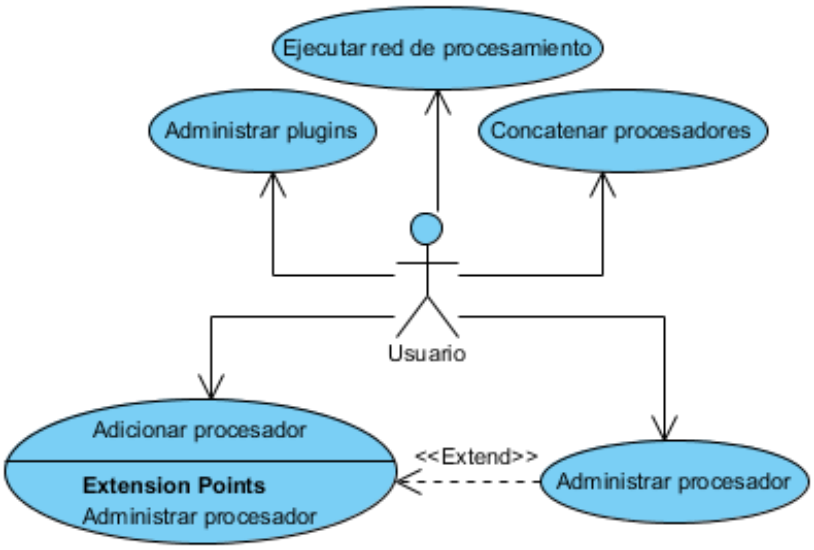

Fig. 20 Vista de casos de uso.

A continuación se describen los elementos que intervienen en esta vista.

2.6.1.1 Actores del sistema

Los actores son las personas, sistemas o hardware externo que interactuará con la aplicación [21]. En este caso quien hará uso del prototipo funcional, que se creará para validar la arquitectura, será un usuario genérico, en esta clasificación entrarán todas las personas que interactúen con la aplicación (ver Tabla 3).

Tabla 3 Actores del sistema.

Actor	Descripción
Usuario	Persona que interactuará con la aplicación que se desarrolle, podrá acceder a todas las funcionalidades presentes en la misma.

2.6.1.2 Descripción de los casos de uso.

A continuación se describen los CUAS teniendo en cuenta las acciones que realizará el sistema como respuesta a las peticiones del usuario.

Tabla 4 Caso de uso Administrar *plugins*.

Caso de uso	Administrar *plugins.*
Objetivo	Permitir al usuario administrar los *plugins* (adicionar, habilitar, deshabilitar o eliminar).
Actores	Usuario.
Resumen	El caso de uso se inicia cuando el usuario presiona el botón “Administrar

	plugins", el sistema muestra una interfaz que permite al usuario realizar la acción deseada (adicionar, habilitar, deshabilitar o eliminar *plugins*).
Complejidad	Alta.
Prioridad	Crítica.
Precondiciones	
Postcondiciones	El sistema queda actualizado de acuerdo a la acción seleccionada. Se adiciona, habilita, deshabilita o elimina un *plugin* de la aplicación.

Flujo de eventos

Flujo básico <Administrar *Plugins*>

No	Actor	Sistema
1.	Selecciona la opción "Administrar plugins" haciendo clic en el botón correspondiente, ubicado en la barra de herramientas.	1.1 Muestra la interfaz "Administrar plugins".
2.	Selecciona una de las siguientes opciones haciendo clic en el botón correspondiente: - Adicionar. - Selecciona el *plugin* que desea eliminar y presiona el botón "Eliminar". - Selecciona el *plugin* que desea deshabilitar y presiona el botón "Deshabilitar". - De los *plugins* que se encuentran deshabilitados, selecciona al que desea habilitar y presiona el botón "Habilitar".	2.1 Realiza una de las siguientes acciones de acuerdo a la operación realizada por el usuario: - Adicionar (ir a la Sección 1: "Adicionar *plugin*"). - Eliminar (ir a la Sección 2: "Eliminar plugin"). - Deshabilitar (ir a la Sección 3: "Deshabilitar *plugin*"). - Habilitar (ir a la Sección 4: "Habilitar *plugin*").
3.		3.1 Termina el caso de uso.

Sección 1: "Adicionar *plugin*"

Flujo básico <Adicionar *plugin*>

	Actor	Sistema
1.		1.1 Muestra un cuadro de dialogo para seleccionar un archivo (solo podrán seleccionarse archivos con la extensión .dll).
2.	Busca en el sistema de archivos el *plugin* que desea adicionar a la aplicación y	2.1 Verifica que el *plugin* seleccionado sea válido para la aplicación.

	selecciona la opción Abrir.	
3.		3.1 Adiciona a la lista de *plugins,* que se muestra en la interfaz "Administrar plugins", el *plugin* seleccionado con sus datos fundamentales (nombre, versión, autor, descripción). 3.2 Si el *plugin* que se adicionó es de tipo Procesador, se adiciona además, a la lista de procesadores que se encuentra en la interfaz principal.
4.		4.1 Regresa al paso dos del flujo básico de eventos "Administrar plugins".
Flujos alternos		
Nº Evento <2a. Selecciona la opción Cancelar>		
	Actor	**Sistema**
2a.	Selecciona la opción Cancelar.	2.1a Regresa al paso dos del flujo básico de eventos "Administrar plugins".
Nº Evento <2.1a. El archivo seleccionado no es compatible con la aplicación>		
2a.		2.1a Muestra el mensaje de error: "El archivo seleccionado no es un plugin válido para la aplicación".
3a.	Presiona el botón Aceptar.	3.1a Regresa al paso dos del flujo básico "Administrar plugins".
Sección 2: "Eliminar *plugin*"		
Flujo básico <Eliminar *plugin*>		
	Actor	**Sistema**
1.		1.1 Muestra el mensaje: "Está seguro que desea eliminar el plugin seleccionado".
2.	Presiona el botón Aceptar.	2.1 Elimina el *plugin* de la lista de plugins que se muestra en la interfaz "Administrar plugins". 2.2 Si el *plugin* que se eliminó es de tipo Procesador, se elimina de la lista de procesadores que se encuentra en la interfaz principal.
3.		3.1 Regresa al paso dos del flujo básico de eventos "Administrar plugins".
Flujos alternos		
Nº Evento <2a. Selecciona la opción Cancelar>		
	Actor	**Sistema**

2a.	Selecciona la opción Cancelar.	2.1a Regresa al paso dos del flujo básico de eventos “Administrar plugins”.
Sección 3: “Deshabilitar *plugin*”		
Flujo básico <Deshabilitar *plugin*>		
	Actor	**Sistema**
1.		1.1 Muestra el mensaje: “Está seguro que desea deshabilitar el plugin seleccionado”.
2.	Presiona el botón Aceptar.	2.1 Deshabilita el *plugin* seleccionado, se resalta el mismo empleando un color rojo en la lista de *plugins* de la interfaz “Administrar Plugin”, para diferenciarlo del resto. 2.2 Si el *plugin* que se deshabilitó es de tipo Procesador, se elimina de la lista de procesadores que se encuentra en la interfaz principal.
3.		3.1 Regresa al paso dos del flujo básico de eventos “Administrar plugins”.
Flujos alternos		
Nº Evento <2a. Selecciona la opción Cancelar>		
	Actor	**Sistema**
2a.	Selecciona la opción Cancelar.	2.1a Regresa al paso dos del flujo básico de eventos “Administrar plugins”.
Sección 4: “Habilitar *plugin*”		
Flujo básico <Habilitar *plugin*>		
	Actor	**Sistema**
1.		1.1 Habilita el *plugin* seleccionado, este vuelve a tomar su color normal en la lista de plugins que aparece en la interfaz “Administrar plugins”. 1.2 Si el *plugin* que se habilitó es de tipo Procesador, se restaura en la lista de procesadores que aparece en la interfaz principal.
2.		2.1 Regresa al paso dos del flujo básico de eventos “Administrar plugins”.

Relaciones	**CU Incluidos**	-
	CU Extendidos	-

Tabla 5 Caso de uso Adicionar procesador.

Caso de uso	Adicionar procesador.	
Objetivo	Permitir al usuario adicionar un procesador a la escena de procesamiento.	
Actores	Usuario.	
Resumen	El caso de uso se inicia cuando el usuario selecciona el procesador que desea adicionar a la escena y lo arrastra a la misma.	
Complejidad	Alta.	
Prioridad	Crítica.	
Precondiciones	Debe existir al menos un plugin de tipo procesador en estado activo.	
Postcondiciones	Se adiciona el procesador a la escena.	
Flujo de eventos		
Flujo básico <Adicionar procesador>		
No	**Actor**	**Sistema**
1.	Selecciona el procesador de la lista de procesadores que se encuentra en la interfaz principal. 1.1 Arrastra el procesador a la escena de procesamiento.	1.2 Adiciona el procesador a la escena de procesamiento.
2.		2.1 Termina el caso de uso.
Relaciones	**CU Incluidos**	-
	CU Extendidos	Administrar procesador.

Tabla 6 Administrar procesador.

Caso de uso	Administrar procesador.
Objetivo	Permitir al usuario modificar las propiedades de configuración de un procesador adicionado a la escena o eliminarlo de la misma.
Actores	Usuario.
Resumen	El caso de uso se inicia cuando el usuario da doble clic sobre un procesador que se encuentra en la escena o clic derecho sobre el mismo.
Complejidad	Alta.

Prioridad	Crítica.
Precondiciones	Debe haberse adicionado al menos un procesador a la escena.
Postcondiciones	Se modifican las propiedades de configuración de un procesador o se elimina el mismo de la escena.

Flujo de eventos

Flujo básico <Administrar procesador>

No	Actor	Sistema
1.	Ejecuta una de las siguientes acciones: - Da doble clic sobre el procesador. - Da clic derecho sobre el procesador y selecciona la opción Eliminar, o presiona la tecla *Delete*.	1.1 Realiza una de las siguientes acciones de acuerdo a la opción selecciona por el usuario: - Si realiza la primera acción ir a la Sección 1: “Modificar propiedades. - Si realiza la segunda acción ir a la Sección 2: “Eliminar procesador”.

Sección 1: “Modificar propiedades”

Flujo básico <Modificar propiedades>

	Actor	Sistema
1.		1.1 Muestra, en caso que el procesador lo requiera, las propiedades del mismo para su configuración.
2.	Modifica las propiedades deseadas. 2.1 Presiona el botón Aceptar.	2.2 Actualiza los cambios realizados.
3.		3.1 Termina el caso de uso.

Flujos alternos

Nº Evento <2.1a. Presiona el botón Cancelar>

	Actor	Sistema
2a.	Presiona el botón Cancelar.	2.1a Muestra la interfaz principal.

Sección 2: “Eliminar procesador”

Flujo básico <Eliminar procesador>

	Actor	Sistema
1.		1.1 Muestra el mensaje: “Está seguro que desea eliminar el procesador de la escena”.
2.	Presiona el botón Aceptar.	2.2 Elimina el procesador de la escena, en caso de este procesador haberse concatenado con otro se eliminará también la conexión.

3.		3.1 Termina el caso de uso.
Flujos alternos		
Nº Evento <2.1a. Presiona el botón Cancelar>		
	Actor	**Sistema**
2a.	Presiona el botón Cancelar.	2.1a Muestra la interfaz principal.

Relaciones	**CU Incluidos**	-
	CU Extendidos	-

Tabla 7 Concatenar procesadores.

Caso de uso	Concatenar procesadores.
Objetivo	Permitir al usuario concatenar dos procesadores.
Actores	Usuario.
Resumen	El caso de uso se inicia cuando el usuario selecciona el puerto de salida de un procesador y el puerto de entrada de otro procesador.
Complejidad	Alta.
Prioridad	Crítica.
Precondiciones	Deben existir al menos dos procesadores en la escena.
Postcondiciones	Se crea una red de procesamiento.
Flujo de eventos	
Flujo básico <Adicionar procesador>	

No	**Actor**	**Sistema**
1.	Selecciona el puerto de salida del procesador que desea concatenar.	1.1 Cambia el color del puerto de salida de verde a azul, hasta que el usuario realice una nueva acción.
2.	Selecciona el puerto de entrada del procesador con el cual desea concatenar al anterior.	2.1 Cambia el color del puerto de entrada de rojo a azul, hasta que el usuario realice una nueva acción.
3.		3.1 Verifica la compatibilidad de los procesadores. 3.2 Dibuja la conexión entre ambos procesadores.
4.		4.1 Termina el caso de uso.

Flujos alternos		
Nº Evento <3.1.a. Los procesadores no son compatibles>		
	Actor	**Sistema**
3a.		3.1a Muestra el mensaje “Procesadores no compatibles”.
4a.	Selecciona la opción Aceptar.	4.1a. Regresa al paso uno del flujo básico de eventos “Concatenar procesadores”.

Relaciones	**CU Incluidos**	-
	CU Extendidos	-

Tabla 8 Ejecutar red de procesamiento.

Caso de uso	Ejecutar red de procesamiento.
Objetivo	Permitir al usuario ejecutar la red de procesamiento que se encuentre en la escena.
Actores	Usuario
Resumen	El caso de uso se inicia cuando el usuario presiona el botón Ejecutar que se encuentra en la barra de herramientas.
Complejidad	Alta
Prioridad	Crítica
Precondiciones	Debe existir al menos un procesador en la escena.
Postcondiciones	Se ejecuta la acción de los procesadores que se encuentren en la red.

Flujo de eventos		
Flujo básico <Ejecutar escena>		
No	**Actor**	**Sistema**
1.	Da clic en el botón Ejecutar que se encuentra en la barra de herramientas.	1.1 Ejecuta la acción contenida en los procesadores que se encuentren en la red.
2.		2.1 Termina el caso de uso.

Relaciones	**CU Incluidos**	-
	CU Extendidos	-

2.6.2 Vista lógica.

La vista lógica comprende los elementos del diseño significativos para la arquitectura. En la Fig. 21 se observan cada una de las capas que posee la propuesta arquitectónica, así como las clases fundamentales que intervienen en la misma.

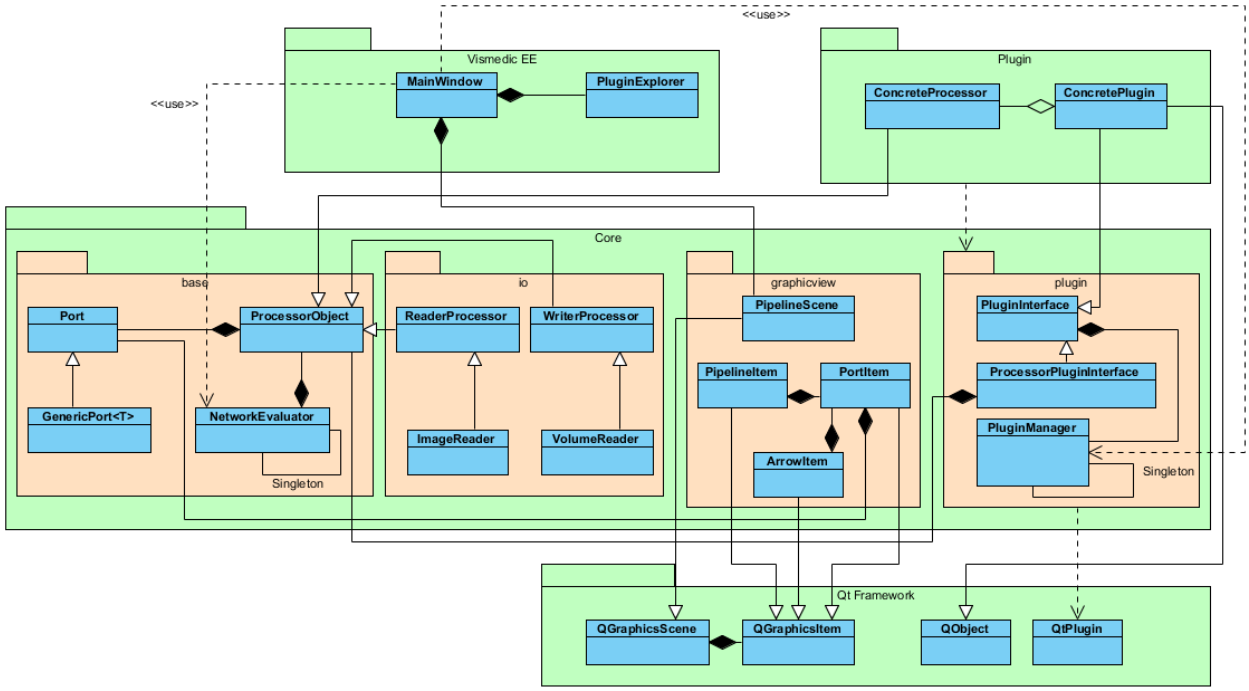

Fig. 21 Vista lógica.

Como se observa en la Fig. 21, el núcleo (**Core**) de la arquitectura se encuentra soportado por las clases del *framework* Qt, haciendo uso extensivo de sus funcionalidades para el manejo de *plugins* y del *Graphics-View Framework* para la representación dinámica de la red de flujo de datos, las clases fundamentales empleadas de este módulo de Qt son: ***QGraphicsScene*** y ***QGraphicsItem***. A continuación se describen las clases principales, que constituyen la base para toda la arquitectura y permiten la composición de la red de flujo de datos (ver Fig. 22).

Port: clase genérica que representa los puntos de conexión entre los objetos de tipo ***ProcessorObject***.

ProcessorObject: clase encargada de procesar (leer, filtrar, escribir, segmentar, visualizar, etc.).

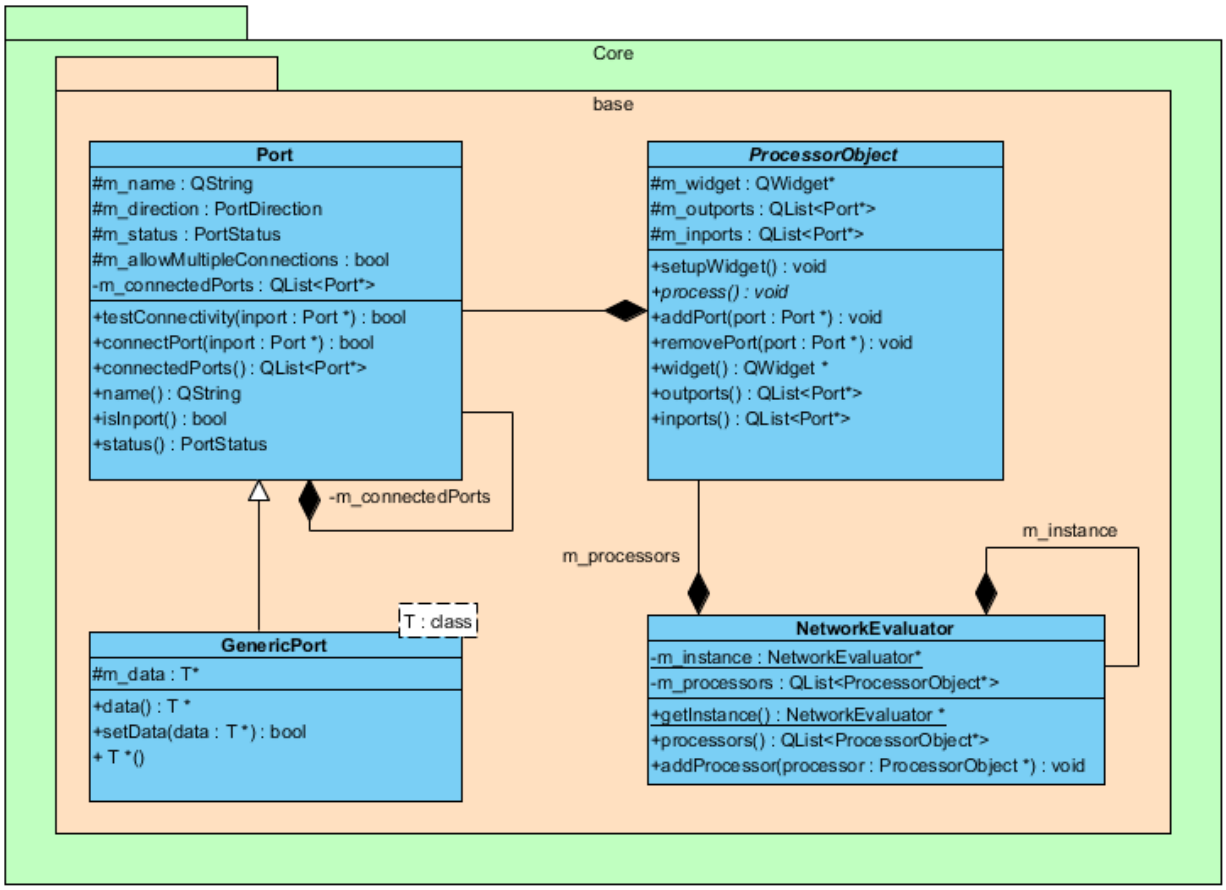

Fig. 22 Clases fundamentales para formar la red de flujo de datos.

Los componentes que soportará la arquitectura deben construirse en forma de *plugins*, desde el punto de vista del desarrollador, esto significa crear un nuevo proyecto y heredar de la interfaz ***PluginInterface***, ubicada en el módulo ***Plugin***. Dentro del *plugin* se implementa el procesador concreto y si es necesario los puertos específicos para este procesador. Para integrar el nuevo *plugin* al entorno de ejecución, solo es necesario copiarlo para la carpeta *plugins* (ubicada en el directorio de instalación). El entorno de ejecución se encargará automáticamente de cargar todos los *plugins,* permitiendo posteriormente la edición y evaluación de la red de flujo de datos de forma dinámica.

Para soportar el versionado de los *plugins*, se adicionó a la interfaz ***PluginInterface*** los atributos: *majorVersion*, *minorVersion* y *microVersion*, de esta forma se garantiza la existencia en tiempo de ejecución de varias versiones de un mismo *plugin*. En la Fig. 23 se observa el diagrama de clases del diseño del módulo *plugin*.

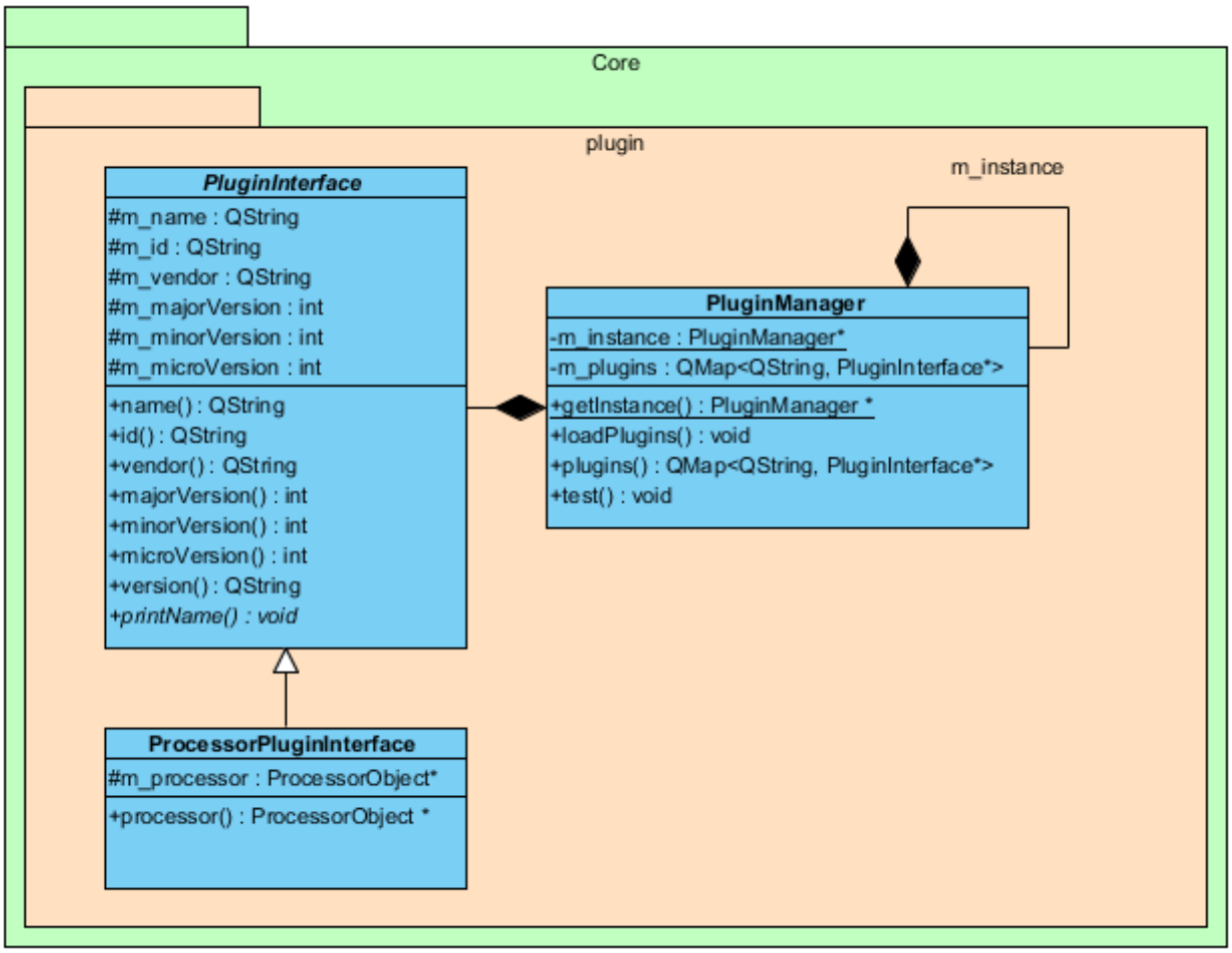

Fig. 23 Clases fundamentales pertenecientes al módulo Plugin.

2.6.3 Vista de implementación.

La vista de implementación muestra los elementos físicos que componen el sistema, por ejemplo: componentes, ficheros y librerías. Se representa a través de un diagrama de componentes y refleja la organización de módulos de software dentro del entorno de desarrollo. En la Fig. 24 se observa la vista de implementación de la arquitectura propuesta.

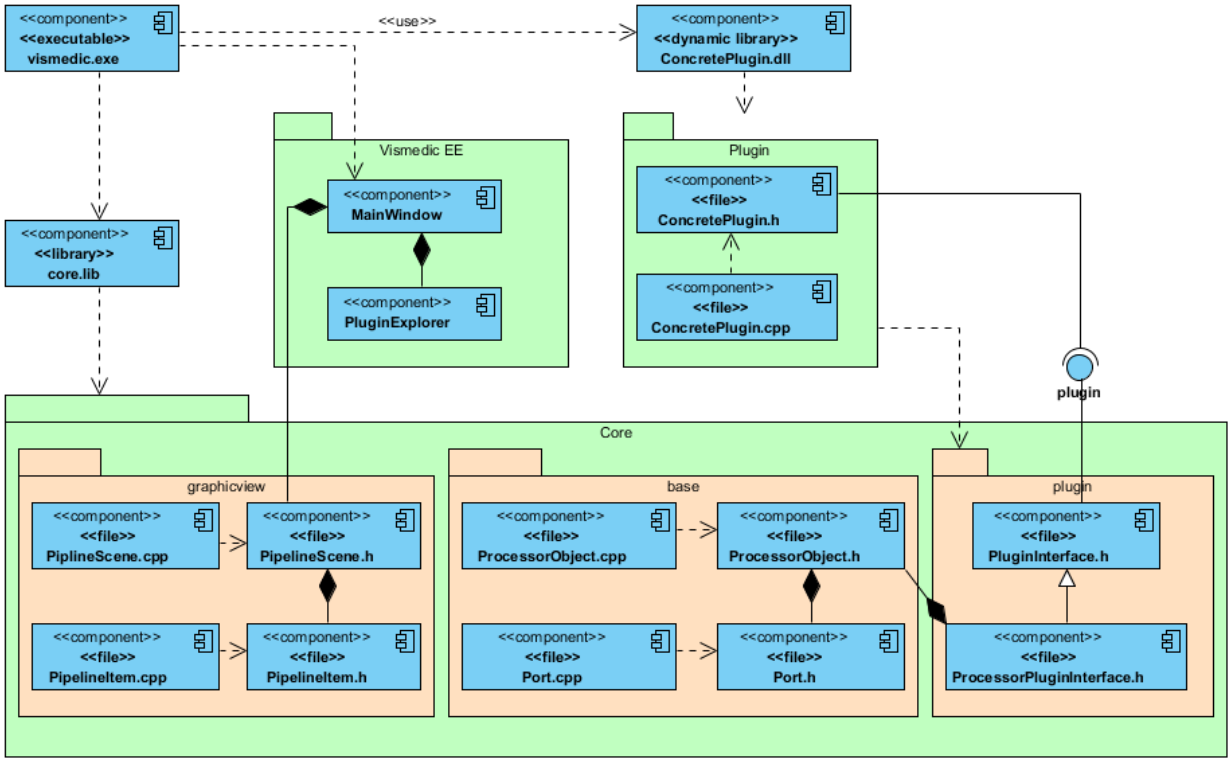

Fig. 24 Vista de implementación.

2.6.4 Vista de despliegue.

La vista de despliegue muestra la configuración de los nodos (procesadores y dispositivos) que participan en la ejecución y de los componentes que residen en los mismos, mostrando la colaboración entre nodos mediante protocolos de comunicación. De acuerdo a la planificación del proyecto, las aplicaciones a desarrollar próximamente, sobre la base de la arquitectura propuesta, serán desplegadas en estaciones de trabajo independientes que cumplan con los requerimientos de hardware y software especificados en la sección 2.4 (ver Fig. 25).

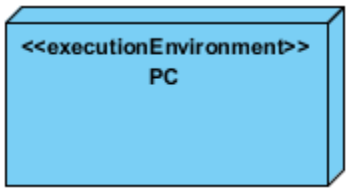

Fig. 25 Vista de despliegue.

2.7 Patrones de diseño utilizados

En el diseño e implementación de la propuesta de solución, se tuvo en cuenta el estudio realizado sobre los patrones de diseño y de asignación general de responsabilidades (ver secciones 1.3 y 1.4), teniendo en cuenta las características de la arquitectura propuesta se utilizaron los patrones *Singleton* y *Factory Method*, además, se empleó el Polimorfismo, perteneciente a los patrones

GRASP. A continuación se muestra cómo se evidencia el uso de estos patrones dentro de la solución.

Singleton

Este patrón se empleó en las clases ***PluginManager*** y ***NetworkEvaluator*** (Fig. 22 y Fig. 23), ambas precisan de una única instancia, independientemente del punto de acceso dentro de la aplicación. Las dos clases implementan el método estático ***getInstance()*** que se encarga de retornar la única instancia de la clase.

Factory Method

Para la adición dinámica de los procesadores a la red de flujo de datos, cada *plugin* que implemente la interfaz ***ProcessorPluginInterface*** debe conocer qué tipo de procesador concreto debe instanciar, para esto se adicionó a la interfaz el método abstracto ***instantiateProcessor()***, la implementación de este método en los *plugins*, permite que se delegue a estos la responsabilidad de crear los procesadores concretos.

Polimorfismo

Este principio se utilizó fundamentalmente en los objetos de tipo procesador, donde, la clase ***ProcessorObject*** declara el método polimórfico ***process()***, que debe ser implementado por todos los procesadores concretos. Para evaluar la red de procesamiento, la clase encargada de ejecutar esta acción (***NetworkEvaluator***) invoca el método ***process()*** de cada uno de los procesadores presentes en la red. El uso del polimorfismo permite determinar en tiempo de ejecución qué implementación concreta del método ***process()*** debe invocarse.

CAPÍTULO 3. EVALUACIÓN DE LA ARQUITECTURA

En este capítulo se exponen algunos conceptos asociados a la evaluación de las arquitecturas de software, fundamentalmente las técnicas y métodos que se emplean, así como los atributos de calidad que se miden. Se evalúa la arquitectura propuesta empleando la técnica basada en prototipos y el Método de Análisis de Acuerdos de Arquitectura de Software.

3.1 Evaluación de la arquitectura de software

La arquitectura de software posee gran impacto sobre la calidad de un sistema, por lo que se hace necesario evaluarla para determinar su potencial para alcanzar los atributos de calidad requeridos. Es importante destacar que la evaluación no define si una arquitectura es buena o no, simplemente expresa donde se encuentran los riesgos y fortalezas de la misma.

El primer paso para evaluar una arquitectura es conocer qué es lo que se quiere evaluar, de esta forma es posible establecer la base para la evaluación, otra decisión importante es determinar cuándo se realizará la evaluación. Para esto, aunque es posible evaluar la arquitectura en cualquier fase del desarrollo, existen dos variantes definidas de cuando realizar la evaluación: evaluación temprana y evaluación tardía [22].

Para realizar la evaluación temprana no es necesario esperar que la arquitectura esté totalmente especificada, esta puede realizarse desde las fases tempranas del diseño y a lo largo del proceso de desarrollo, lo que permite tomar decisiones arquitectónicas producto a una evaluación en función de los atributos de calidad esperados. La evaluación tardía se realiza cuando la arquitectura ya está establecida y la implementación se ha culminado, realizar la evaluación en este momento se considera muy útil, pues se puede observar el cumplimiento de los atributos de calidad asociados al sistema y su comportamiento de forma general [22].

3.1.1 Atributos de calidad

Los atributos de calidad, a partir de los cuales se evalúa la arquitectura, son requerimientos del sistema que hacen referencia a las características que éste debe satisfacer. Estos atributos se clasifican en dos grupos: observables vía ejecución (OVE) y no observables vía ejecución (NOVE). Los atributos que se clasifican como OVE se determinan del comportamiento del sistema en tiempo de ejecución y los NOVE se establecen durante el desarrollo del sistema. La descripción de algunos de estos atributos se encuentra en la Tabla 9.

Tabla 9 Atributos de calidad del software.

Atributo de calidad	Categoría	Descripción
Confiabilidad	OVE	Medida de la habilidad de un sistema de mantenerse operativo a lo largo del tiempo [23].

Confidencialidad	OVE	Ausencia de acceso no autorizado a la información [23].
Configurabilidad	NOVE	Posibilidad que se otorga a un usuario experto de realizar ciertos cambios al sistema [24].
Desempeño	OVE	Grado en el cual un sistema o componente cumple con las funciones designadas dentro de ciertas restricciones dadas [25].
Disponibilidad	OVE	Medida de disponibilidad del sistema para el uso [23].
Escalabilidad	NOVE	Grado con el que se puede ampliar el diseño arquitectónico, de datos o procedimental [26].
Funcionalidad	OVE	Habilidad del sistema para realizar el trabajo para el que fue realizado [22].
Integrabilidad	NOVE	Medida en que trabajan correctamente componentes del sistema que fueron desarrollados separadamente al ser integrados [6].
Mantenibilidad	NOVE	Capacidad de modificar el sistema de manera rápida y a bajo costo [24].
Modificabilidad	NOVE	Habilidad de realizar cambios futuros al sistema [24].
Portabilidad	NOVE	Habilidad del sistema para ser ejecutado en diferentes plataformas. Estos ambientes pueden ser hardware, software o una combinación de los dos [22].
Reusabilidad	NOVE	Es la capacidad de diseñar un sistema de forma tal que su estructura o parte de sus componentes puedan ser reutilizados en futuras aplicaciones [6].

Los atributos de calidad se ven altamente influenciados por la arquitectura del sistema, aunque existen atributos que no dependen directamente de la arquitectura (ejemplo: usabilidad). Es por esto que la calidad del sistema debe ser considerada en todas las fases del diseño.

Es importante tener en cuenta que no puede lograrse la satisfacción de ciertos atributos de calidad de manera aislada, satisfacer un atributo de calidad puede tener efectos positivos o negativos sobre otros atributos que, de alguna manera, también se desean alcanzar [6].

3.1.2 Modelos de calidad de software

Los atributos de calidad se pueden organizar y descomponer de maneras diferentes, en lo que se conoce como modelos de calidad de software (MCS), estos modelos facilitan el entendimiento del proceso de ingeniería de software [27]. Algunos de los MCS propuestos hasta el momento

son:

- Modelo de McCall (1977).
- Modelo de FURPS (1987).
- Modelo ISO/IEC 9126 (1991).
- Modelo de Dromey (1996).
- Modelo ISO/IEC 9126 adaptado para arquitecturas de software (2003).

El modelo ISO/IEC 9126 adaptado para arquitecturas de software, como su nombre lo indica, es una adaptación del modelo ISO/IEC 9126 y fue propuesto en el año 2003 por Losavio et al [28]. Este modelo se basa en los atributos de calidad que se relacionan directamente con la arquitectura: funcionalidad, confiabilidad, eficiencia, mantenibilidad y portabilidad. No se considera como parte de este modelo el atributo usabilidad, propuesto por el modelo original, pues se relaciona con los componentes de la interfaz gráfica de usuario y estos son independientes de la arquitectura.

3.2 Técnicas de evaluación de arquitecturas

Las técnicas de evaluación de arquitecturas tienen como principal objetivo, crear especificaciones y predicciones respecto a una propuesta arquitectónica, para saber si satisface las cualidades que el software debe cumplir. Posibilitan además, establecer comparaciones entre arquitecturas candidatas para determinar cuál satisface mejor un atributo de calidad específico. Estas técnicas pueden clasificarse en dos vertientes fundamentales: cuantitativas y cualitativas (ver Fig. 26).

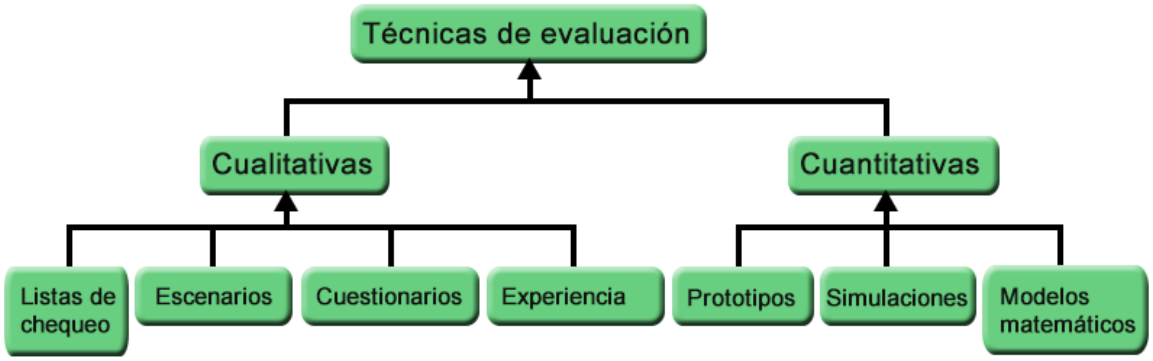

Fig. 26 Técnicas de evaluación.

Las técnicas de evaluación cualitativas son usadas cuando la arquitectura se encuentra en construcción, a diferencia de las cuantitativas, que se utilizan cuando la arquitectura ya ha sido implantada. Es más común el uso de las técnicas cualitativas, pues permiten tomar decisiones arquitectónicas en fases tempranas del desarrollo, requieren menor información detallada y pueden conducir a resultados relativamente precisos, además, son menos costosas de realizar que las cuantitativas [29].

A continuación se hace un análisis de las técnicas más propicias a ser empleadas para la evaluación de la arquitectura propuesta.

3.2.1 Evaluación basada en escenarios

Un escenario es una breve descripción de la interacción de alguno de los involucrados en el desarrollo del sistema con éste, dígase usuarios o desarrolladores, por ejemplo. Consta de tres partes: el estímulo, el contexto y la respuesta. El estímulo es la parte del escenario que explica o describe lo que el involucrado en el desarrollo hace para iniciar la interacción con el sistema, puede incluir ejecución de tareas, cambios en el sistema, ejecución de pruebas, configuración, entre otros. El contexto describe qué sucede en el sistema al momento del estímulo. La respuesta describe, a través de la arquitectura, cómo debería responder el sistema ante el estímulo. Este último elemento permite establecer cuál es el atributo de calidad asociado [22]. Esta técnica cuenta con instrumentos de evaluación, donde se destacan el Árbol de Utilidad (*Utility Tree*) y los Perfiles (*Profiles*).

El **Árbol de Utilidad** es un esquema en forma de árbol que presenta los atributos de calidad de un sistema de software, refinados hasta el establecimiento de escenarios que especifican el nivel de prioridad de un atributo de calidad. Su intención es identificar los atributos de calidad más importantes para el proyecto. No existe un conjunto preestablecido de atributos, sino que son definidos por los involucrados en el desarrollo del sistema al momento de la construcción del árbol [22].

Un **Perfil** es un conjunto de escenarios, con alguna importancia relativa asociada a cada uno de ellos. El uso de perfiles permite hacer especificaciones más precisas del requerimiento para un atributo de calidad. Los perfiles tienen asociados dos formas de especificación: perfiles completos y perfiles seleccionados. Definir un perfil consta de tres pasos: definir las categorías del escenario, seleccionar y definir los escenarios para la categoría y asignar el peso a los escenarios. Cada atributo de calidad va a tener asociado un perfil [29].

Los perfiles completos definen todos los escenarios relevantes como parte del perfil. Esto permite realizar un análisis de la arquitectura para un atributo de calidad de manera completa, pues incluye todos los posibles casos. Su uso se reduce a sistemas relativamente pequeños y sólo es posible predecir conjuntos de escenarios completos para algunos atributos de calidad. Los perfiles seleccionados, consisten en cambio, en tomar un conjunto de escenarios de forma aleatoria, de acuerdo con algunos requerimientos [29].

Entre las principales ventajas de esta técnica se destaca que: los escenarios son fáciles de crear y entender, el tiempo de aplicación es relativamente corto, la experiencia que se necesita para aplicar la técnica es poca y la misma brinda facilidad de evaluación de características de calidad, observables o no en tiempo de ejecución. Entre los aspectos no favorables con que cuenta la técnica se encuentra su subjetividad [29].

3.2.2 Evaluación basada en la experiencia

En muchas ocasiones los arquitectos e ingenieros de software otorgan valiosas ideas que resultan de utilidad para la evasión de decisiones erradas de diseño, estas ideas se basan en factores subjetivos (como la experiencia) [29] y están respaldadas por una línea lógica de razonamiento, que se puede adquirir por el trabajo realizado en proyectos similares. Por tanto, el principal instrumento de evaluación con que cuenta esta técnica es precisamente la intuición y experiencia con que cuentan los arquitectos y demás miembros del equipo de desarrollo. Existen dos tipos de evaluación basada en experiencia: la evaluación informal, que es realizada por los arquitectos de software durante el proceso de diseño, y la realizada por equipos externos de evaluación de arquitecturas.

3.2.3 Evaluación basada en simulación

La evaluación basada en simulación utiliza una implementación de alto nivel de la arquitectura de software. El enfoque básico consiste en la implementación de componentes de la arquitectura y la implementación, a cierto nivel de abstracción, del contexto del sistema donde se supone va a ejecutarse. Su finalidad es evaluar el comportamiento de la arquitectura bajo diversas circunstancias, una vez disponibles estas implementaciones, pueden usarse los perfiles respectivos para evaluar los atributos de calidad.

La exactitud de los resultados de la evaluación depende, a su vez, de la exactitud del perfil utilizado para evaluar el atributo de calidad y de la precisión con la que el contexto del sistema simula las condiciones del mundo real. Los instrumentos asociados a esta técnica son los lenguajes de descripción arquitectónica (ADL[4] por sus siglas en inglés) y los modelos de colas, siendo esto una de sus desventajas, pues el evaluador debe tener experiencia en el uso de los ADL o los modelos de cola para realizar una correcta evaluación [29].

3.2.4 Evaluación basada en prototipo

Esta técnica consiste en implementar una parte de la arquitectura de software y ejecutarla en el contexto del sistema. Para su uso se necesita mayor información sobre el desarrollo y disponibilidad de hardware, y los elementos que constituyen el contexto del sistema de software. Mediante esta técnica se obtiene un resultado de evaluación con mayor exactitud [29].

Entre los aspectos favorables del uso de esta técnica se destacan la confiabilidad, pues se puede ver de manera directa que tanto se afecta o no, un atributo de calidad en el sistema que se desarrolla, los usuarios finales pueden observar el resultado que se está obteniendo y evaluar junto al equipo de desarrollo si satisface o no sus expectativas. De acuerdo al nivel de fidelidad con que se desarrolle el prototipo del producto final, puede suponer desventajas el empleo de esta técnica, pues el tiempo de desarrollo del prototipo puede ser largo (lo que convierte a la

[4] ADL: *Architecture Description Languages.*

técnica en costosa) y demorar en poder realizar la evaluación.

3.3 Métodos de evaluación de arquitecturas

Los métodos de evaluación arquitectónica, evalúan el potencial del diseño arquitectónico para alcanzar los niveles deseados en cuanto a los requisitos de calidad [6], estos métodos se ven asistidos de las técnicas de evaluación arquitectónica analizadas anteriormente. Actualmente existen diversos métodos para realizar pruebas a la arquitectura de software, cada uno con características específicas, escoger un método de evaluación requiere tener bien definidos los atributos que se desean evaluar. Algunos de los métodos de evaluación son los siguientes:

- Método de Análisis de Arquitectura de Software (*Software Architecture Analysis Method,* SAAM).
- Método de Revisión Intermedio de Diseño (*Active Reviews for Intermediate Designs*, ARID).
- Método de Análisis de Acuerdos de Arquitectura de Software (*Architecture Trade-off Analysis Method*, ATAM).

3.3.1 SAAM

El SAAM fue el primer método en ser ampliamente difundido y documentado. Se creó originalmente para el análisis de la modificabilidad de una arquitectura, pero en la práctica ha demostrado ser muy útil para evaluar de forma rápida atributos de calidad, tales como la portabilidad, escalabilidad e integrabilidad [22].

Este método se enfoca en la enumeración de un conjunto de escenarios que representan los cambios probables a los que estará sometido el sistema. Si el objetivo de aplicar el método es evaluar una sola arquitectura, se obtienen los lugares en los que la misma puede fallar, en términos de los requerimientos de modificabilidad. Para el caso en el que se cuenta con varias arquitecturas candidatas, el método produce una escala relativa que permite observar qué opción satisface mejor los requerimientos de calidad con la menor cantidad de modificaciones. La Tabla 10 presenta los pasos que contempla el SAAM, con una breve descripción de los mismos [30].

Tabla 10 Pasos contemplados por SAAM.

Pasos del SAAM	Descripción
1. Desarrollo de escenarios	Un escenario es una breve descripción de usos anticipados o deseados del sistema. De igual forma, estos pueden incluir cambios a los que puede estar expuesto el sistema en el futuro.
2. Descripción de la arquitectura	La arquitectura (o las arquitecturas candidatas) debe ser descrita haciendo uso de alguna notación arquitectónica

	que sea común a todas las partes involucradas en el análisis. Deben incluirse los componentes de datos y conexiones relevantes, así como la descripción del comportamiento general del sistema. El desarrollo de escenarios y la descripción de la arquitectura son usualmente llevados a cabo de forma intercalada, o a través de varias iteraciones.
3. Clasificación y asignación de prioridad de los escenarios	La clasificación de los escenarios puede hacerse en dos clases: directos e indirectos. Un escenario directo es el que puede satisfacerse sin la necesidad de modificaciones en la arquitectura. Un escenario indirecto es el que requiere modificaciones en la arquitectura para poder satisfacerse. Los escenarios indirectos son de especial interés para SAAM, pues permiten medir el grado en el que una arquitectura puede ajustarse a los cambios de evolución que son importantes para los involucrados en el desarrollo.
4. Evaluación individual de los escenarios indirectos	Para cada escenario indirecto, se listan los cambios necesarios sobre la arquitectura, y se calcula su costo. Una modificación sobre la arquitectura significa que debe introducirse un nuevo componente o conector, o que alguno de los existentes requiere cambios en su especificación.
5. Evaluación de la interacción entre escenarios	Cuando dos o más escenarios indirectos proponen cambios sobre un mismo componente, se dice que interactúan sobre ese componente. De forma similar, puede verificarse si la arquitectura se encuentra documentada a un nivel correcto de descomposición estructural.
6. Creación de la evaluación global	Debe asignársele un peso a cada escenario, en términos de su importancia relativa al éxito del sistema. Esta asignación de peso suele hacerse con base en las metas del negocio que cada escenario soporta. En el caso de la evaluación de múltiples arquitecturas, la asignación de pesos puede ser utilizada para la determinación de una escala general.

3.3.2 ARID

El ARID es conveniente para realizar la evaluación de diseños parciales en las etapas tempranas del desarrollo. Es un híbrido entre *Active Design Review* (ADR) y ATAM. Las preguntas giran en torno a la calidad y completitud de la documentación y la suficiencia, ajuste y conveniencia de los servicios que provee el diseño propuesto [22]. ARID consta de dos fases y nueve pasos para la evaluación de la arquitectura, estos se muestran en la Tabla 11 [30].

Tabla 11 Pasos propuestos por el ARID.

Pasos del ARID	Descripción
Fase 1: Actividades previas	
1. Identificación de los encargados de la revisión	Los encargados de la revisión son los ingenieros de software que se espera que usen el diseño, y todos los involucrados en el diseño.
2. Preparar el informe de diseño	El diseñador prepara un informe que explica el diseño. Se incluyen ejemplos del uso del mismo para la resolución de problemas reales. Esto permite al facilitador anticipar el tipo de preguntas posibles, así como identificar áreas en las que la presentación puede ser mejorada
3. Preparar los escenarios base	El diseñador y el facilitador preparan un conjunto de escenarios base, que pueden o no ser utilizados para efectos de la evaluación.
4. Preparar los materiales	Se reproducen los materiales preparados para ser presentados en la segunda fase. Se establece la reunión, y los involucrados son invitados.
Fase 2: Revisión	
5. Presentación del ARID	Se explican los pasos del ARID a los participantes.
6. . Presentación del diseño	El líder del equipo de diseño realiza una presentación, con ejemplos incluidos. Se propone evitar preguntas que conciernen a la implementación o argumentación, así como alternativas de diseño. El objetivo es verificar que el diseño es conveniente.
7. Lluvia de ideas y establecimiento de prioridad de escenarios	Se establece una sesión para la lluvia de ideas sobre los escenarios y el establecimiento de prioridad de escenarios. Los involucrados proponen escenarios a ser usados en el diseño para resolver problemas que esperan

	encontrar. Luego, los escenarios son sometidos a votación, y se utilizan los que resultan ganadores para hacer pruebas sobre el diseño.
8. Aplicación de los escenarios	Comenzando con el escenario que contó con más votos, el facilitador solicita el pseudo-código que utiliza el diseño para proveer el servicio. Este paso continúa hasta que ocurra alguno de los siguientes eventos: - Se agota el tiempo destinado a la revisión. - Se han estudiado los escenarios de más alta prioridad. - El grupo se siente satisfecho con la conclusión alcanzada. Puede suceder que el diseño presentado sea conveniente, con la exitosa aplicación de los escenarios, o por el contrario, no conveniente, cuando el grupo encuentra problemas o deficiencias
9. Resumen	El facilitador recuenta la lista de puntos tratados y pide opiniones a los participantes sobre la eficiencia del ejercicio de revisión.

3.3.3 ATAM

El ATAM está inspirado en tres áreas distintas: los estilos arquitectónicos, el análisis de atributos de calidad y el método de evaluación SAAM [22]. El método se concentra en la identificación de los estilos arquitectónicos o enfoques arquitectónicos utilizados y ayuda a los involucrados en el proyecto a entender las consecuencias de las decisiones arquitectónicas tomadas con respecto a los atributos de calidad. Su desarrollo se basa nueve pasos agrupados en cuatro fases. La Tabla 12 muestra los pasos del método, se encuentra dividida por las fases y contiene una descripción de estos.

Tabla 12 Pasos del ATAM.

Pasos del ATAM	Descripción
Fase 1: Presentación	
1. Presentación del ATAM	El líder de evaluación describe el método a los participantes, trata de establecer las expectativas y responde las preguntas propuestas.
2. Presentación de las metas del negocio	Se realiza la descripción de las metas del negocio que motivan el esfuerzo y aclara que se persiguen objetivos

	de tipo arquitectónico.
3. Presentación de la arquitectura	El arquitecto describe la arquitectura, enfocándose en cómo ésta cumple con los objetivos del negocio.
Fase 2: Investigación y análisis	
4. Identificación de los enfoques arquitectónicos.	Se detectan los enfoques arquitectónicos pero no se analizan.
5. Generación del árbol de utilidad	Se especifican, en forma de escenarios, los atributos de calidad que engloban la “utilidad” del sistema. Se anotan los estímulos y respuestas, y se establece la prioridad entre ellos.
6. Análisis de los enfoques arquitectónicos	Con base en los resultados del establecimiento de prioridades del paso anterior, se analizan los elementos del paso 4. En este paso se identifican riesgos arquitectónicos, puntos de sensibilidad y puntos de balance
Fase 3: Pruebas	
7. Lluvia de ideas y establecimiento de la prioridad de los escenarios	Con la colaboración de todos los involucrados, se complementa el conjunto de escenarios.
8. Análisis de los enfoques arquitectónicos	Se repiten las actividades del paso 6, haciendo uso de los resultados del paso 7. Los escenarios son considerados como casos de prueba para confirmar el análisis realizado hasta el momento.
Fase 4: Reporte	
9. Presentación de los resultados	A partir de la información recolectada a lo largo de la evaluación del ATAM, se presentan los resultados a los participantes.

3.4 Evaluación de la arquitectura propuesta

Después del análisis realizado sobre las diferentes técnicas y métodos de evaluación de arquitecturas, se seleccionan, para evaluar la arquitectura propuesta, la técnica de evaluación basada en prototipo y el método de evaluación ATAM.

3.4.1 Evaluación mediante ATAM

El ATAM, como se explicó anteriormente, está compuesto por nueve pasos divididos en cuatro

fases, los cuales se adaptarán de acuerdo a los requerimientos del proyecto. Después de la descripción de la solución propuesta y el análisis realizado en los capítulos 1 y 2, se procede a evaluar la arquitectura propuesta empleando este método.

3.4.1.1 Árbol de Utilidad

La Tabla 13 muestra el árbol de utilidad correspondiente al paso 5 de la fase 2 de ATAM.

Tabla 13 Árbol de utilidad.

Atributo	Subcaracterística	Escenario	Prioridad
Funcionalidad	Adecuación Precisión	Adicionar *plugin.*	Alta
Integrabilidad			
Funcionalidad	Adecuación Precisión	Eliminar *plugin.* Habilitar *plugin.* Deshabilitar *plugin.*	Alta
Confiabilidad	Recuperabilidad	Estabilidad del sistema ante la ocurrencia de errores.	Media
Mantenibilidad	Facilidad de adaptación al cambio.	Adición de nuevos procesadores o modificación de los existentes.	Alta
Portabilidad	Adaptabilidad	Ejecución del sistema en diferentes sistemas operativos.	Media
Reusabilidad-Extensibilidad		Implementación de nuevas funcionalidades a partir de las interfaces existentes.	Alta
Escalabilidad		Nivel de adaptación del sistema al incremento de la cantidad de *plugins*.	Media

Para establecer la prioridad de los escenarios se analizó el riesgo de no contar con la característica que representa el escenario en el sistema, para esto formularon las siguientes interrogantes:

- ¿Qué ocurre si el escenario no se cumple?
- ¿Cuánto esfuerzo es necesario para cumplir el escenario?
- ¿Es posible compensar el no responder a este escenario?

3.4.1.2 Especificación de los escenarios

Las tablas siguientes muestran las descripciones de los escenarios que se especificaron en el árbol de utilidad representado anteriormente.

Tabla 14 Escenario #1.

Escenario #1: Adicionar *plugin.*	
Atributo(s)	Funcionalidad: Adecuación – Precisión. Integrabilidad.
Estímulo	Se selecciona la opción “Adicionar” en la interfaz “Administrar plugins”.
Respuesta	Se adiciona un nuevo *plugin* a la aplicación.
Explicación	El sistema carga el *plugin* seleccionado en la aplicación, siempre que este sea compatible. Se adiciona el mismo a la aplicación de acuerdo a la interfaz que implemente y se integra de forma correcta con los existentes. Además debe adicionarse al directorio **plugins**, ubicado en la carpeta de instalación del sistema, el archivo correspondiente. De no ser compatible muestra un mensaje de error al usuario.

Tabla 15 Escenario #2.

Escenario #2: Eliminar plugin.	
Atributo(s)	Funcionalidad: Adecuación – Precisión.
Estímulo	En la interfaz “Administrar plugins” se selecciona el *plugin* que se desea eliminar y se presiona la opción “Eliminar”.
Respuesta	Se elimina el *plugin* seleccionado del sistema.
Explicación	El sistema debe pedir al usuario una confirmación de la acción que desea realizar. En caso afirmativo debe eliminar el *plugin* seleccionado de todos los lugares donde se utilice en la aplicación, además debe eliminarse del directorio **plugins**, ubicado en la carpeta de instalación del sistema, el archivo correspondiente.

Tabla 16 Escenario #3.

Escenario #3: Deshabilitar *plugin*.	
Atributo(s)	Funcionalidad: Adecuación – Precisión.

Estímulo	En la interfaz “Administrar plugin” se selecciona el *plugin* que se desea deshabilitar y se presiona la opción “Deshabilitar”.
Respuesta	Se deshabilita el *plugin* seleccionado.
Explicación	El sistema debe pedir confirmación al usuario de la acción que desea realizar, en caso afirmativo debe deshabilitar, en todos los lugares donde se utilice, el *plugin* seleccionado. Además debe adicionar el nombre del *plugin* al fichero que contiene los nombres de los *plugins* deshabilitados. Los *plugins* deshabilitados ser resaltarán con color de fondo rojo en la interfaz “Administrar plugins”.

Tabla 17 Escenario #4.

Escenario #4: Habilitar *plugin*.	
Atributo(s)	Funcionalidad: Adecuación – Precisión.
Estímulo	En la interfaz “Administrar plugins” se selecciona un plugin que se encuentra deshabilitado y se presiona la opción “Habilitar”.
Respuesta	El sistema debe habilitar el *plugin* seleccionado.
Explicación	El sistema habilita el *plugin* seleccionado en la aplicación. Se elimina el nombre del *plugin* del fichero que contiene los nombres de los *plugins* deshabilitados.

Tabla 18 Escenario #5.

Escenario #5: Estabilidad del sistema ante la ocurrencia de errores.	
Atributo(s)	Confiabilidad: Recuperabilidad.
Estímulo	Se aplica un cierre forzado a la aplicación.
Respuesta	El sistema no es capaz de recuperar el último estado en el que se encontraba, por lo que se comporta de manera normal.
Explicación	En la arquitectura no existe un mecanismo para recuperar estados ante la ocurrencia de errores críticos. Esto causa que el usuario pueda perder la configuración de la red que tenía en un instante de tiempo determinado.

Tabla 19 Escenario #6.

Escenario #6: Adición de nuevos procesadores o modificación de los existentes.	
Atributo(s)	Mantenibilidad: Facilidad de adaptación al cambio.

Estímulo	Se adicionan nuevos *plugins* de tipo procesador o se sobrescriben los existentes en el directorio **plugins**, ubicado en la carpeta de instalación del sistema.
Respuesta	Al iniciar la aplicación se cargan los *plugins* adicionados.
Explicación	Si los nuevos *plugins* son compatibles con la aplicación, estos son cargados cuando la aplicación se inicie nuevamente. En el caso de los *plugins* que han sido actualizados, se cargan con las actualizaciones incorporadas.

Tabla 20 Escenario #7.

Escenario #7: Ejecución del sistema en diferentes sistemas operativos.	
Atributo(s)	Portabilidad: Adaptabilidad.
Estímulo	Se ejecuta la aplicación en el sistema operativo Ubuntu 13.04 y Windows 8.
Respuesta	La aplicación funciona correctamente.
Explicación	La arquitectura propuesta solo depende del *framework* Qt, el mismo es multiplataforma, lo que permite que la arquitectura pueda emplearse en todas las plataformas que soporta este *framework*.

Tabla 21 Escenario #8.

Escenario #8: Implementación de nuevas funcionalidades a partir de las interfaces existentes.	
Atributo(s)	Reusabilidad-Extensibilidad.
Estímulo	El desarrollador crea un nuevo *plugin* de tipo procesador a partir de la interfaz ***ProcessorObject*** y ***ProcessorPluginInterface***.
Respuesta	El *plugin* creado puede ser empleado para extender las funcionalidades de la aplicación.
Explicación	Como el *plugin* creado implementa las interfaces definidas, este puede ser empleado para extender las funcionalidades de la aplicación sin necesidad de modificar el código existente.

Tabla 22 Escenario #9.

Escenario #9: Nivel de adaptación del sistema al incremento de la cantidad de plugins.	
Atributo(s)	Escalabilidad.

Estímulo	Incorporación de nuevos *plugins* al sistema.
Respuesta	Se adicionan los nuevos plugins a la aplicación mientras no excedan el consumo de memoria permisible por el sistema operativo.
Explicación	El sistema carga los plugins en tiempo de ejecución, cada *plugin* representa un incremento en el consumo de memoria de la aplicación. El consumo excesivo de memoria puede provocar errores críticos.

3.4.1.3 Resultados de la evaluación con ATAM

A partir de la información obtenida tras la aplicación del ATAM, se identificaron tres riesgos en la arquitectura propuesta, estos riesgos están asociados a los escenarios 5, 6 y 9 y se observan en la Tabla 23.

Tabla 23 Riesgos detectados en la arquitectura propuesta.

Escenarios	Riesgos
5	El sistema no es capaz de regresar al último estado estable ante la ocurrencia de un error crítico.
6	Al ubicar directamente los procesadores en el directorio **plugins**, ubicado en la carpeta de instalación, el sistema no notifica, de forma visual, la no incorporación a la aplicación de los procesadores no compatibles con la misma.
9	El incremento no controlado del número de *plugins* en la aplicación, puede provocar un consumo excesivo de memoria, esto puede causar disminución del rendimiento de la aplicación y en el peor de los casos, errores críticos que lleven al cierre inesperado del sistema.

Luego del análisis de los riesgos detectados se determinó que, en el caso de los riesgos asociados a los escenarios 5 y 6, estos no afectan considerablemente el desempeño de las aplicaciones que se pueden desarrollar con la arquitectura propuesta y la probabilidad de ocurrencia del riesgo asociado al escenario 9 es muy baja, por tanto, se decide mitigar los mismos en una segunda iteración de la arquitectura.

3.4.2 Técnica de evaluación: Prototipo

Para la validación de la arquitectura propuesta, además de evaluar la misma usando el ATAM, se confeccionó un prototipo funcional que contiene los CUAS identificados en la sección 2.6.1. Las imágenes del prototipo obtenido se muestran en las figuras 26 y 27. En la Fig. 27 se observa la interfaz “Administrar plugins” y las operaciones que se pueden realizar sobre la misma, dígase Adicionar, Eliminar, Habilitar o Deshabilitar un *plugin.* En la Fig. 28 se observa el resultado de adicionar tres procesadores a la escena de procesamiento, concatenar los mismos y ejecutar la

red de procesamiento creada.

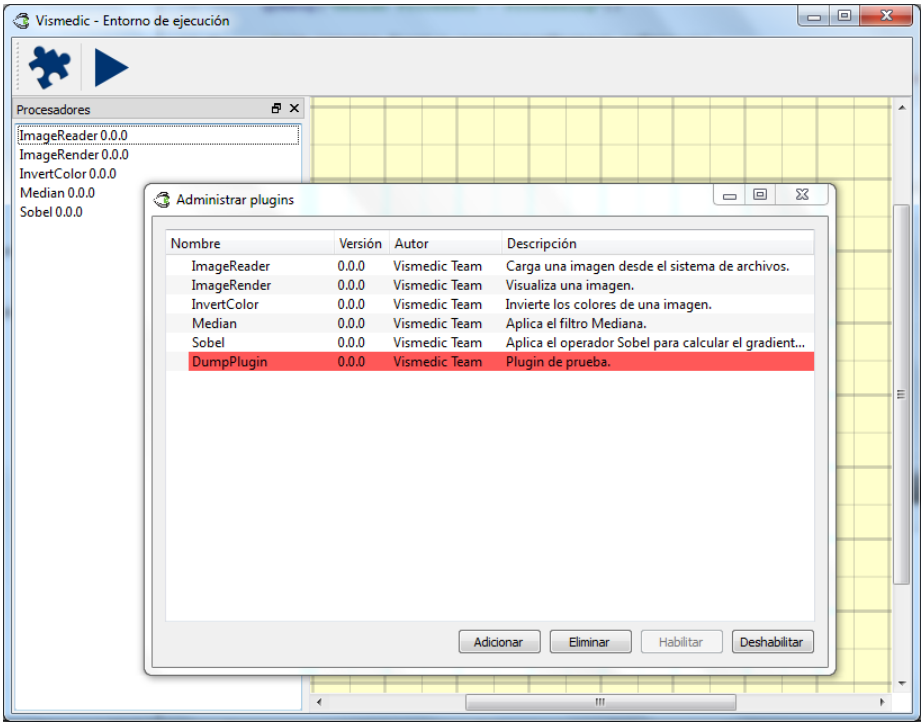

Fig. 27 Prototipo funcional – Administrar plugins.

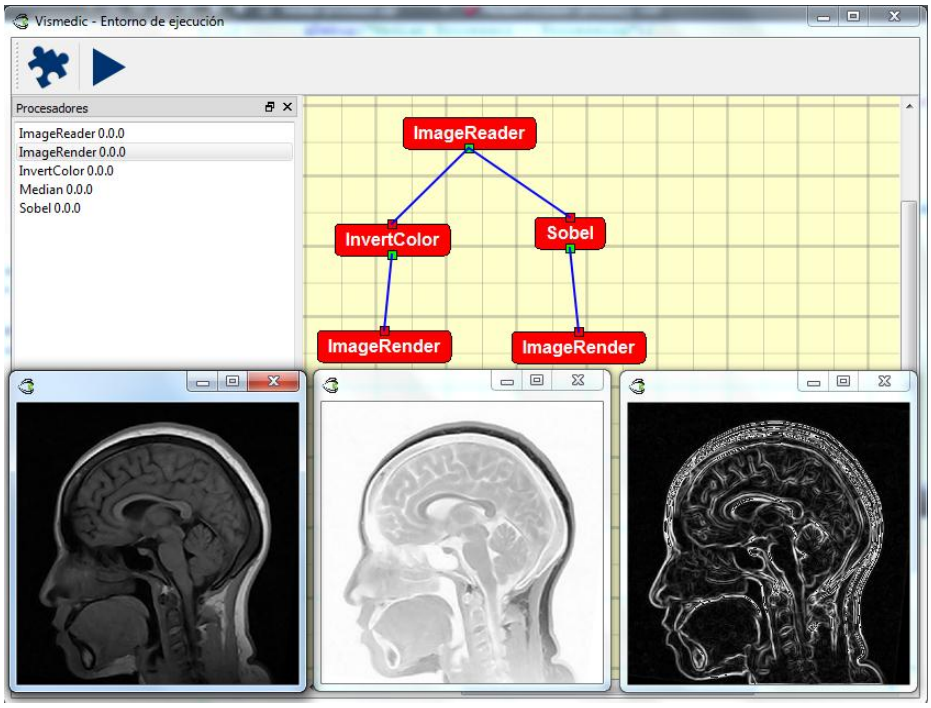

Fig. 28 Prototipo funcional – Concatenar procesadores.

La creación de este prototipo permitió evaluar, de forma práctica, los atributos de calidad de interés para los usuarios, tal es el caso de la integrabilidad entre los componentes que fueron desarrollados de forma separada al integrarse a una misma aplicación y la reusabilidad y extensibilidad que se obtiene al crear los procesadores en forma de *plugins*. De esta forma la arquitectura propuesta satisface los principales atributos de calidad definidos para la investigación.

3.5 Discusión general sobre la arquitectura propuesta

Luego de la aplicación del ATAM y la técnica de evaluación basada en prototipo, se comprobó que el empleo de la arquitectura propuesta influye positivamente en los atributos de calidad que se ven afectados con el uso de la arquitectura actual (ver sección 1.7). A continuación se especifica el efecto de la aplicación de la arquitectura propuesta sobre los principales atributos de calidad:

Portabilidad: la arquitectura propuesta solo depende del *framework* Qt, una de las principales características de este *framework* es su portabilidad hacia diferentes plataformas, tanto de escritorio como móviles. El empleo de una biblioteca externa como DCMTK se delega hacia la implementación de los plugins concretos, por lo que si esta biblioteca presenta problemas de portabilidad solo se afecta el plugin que la emplea. Se recomienda para la lectura de las imágenes DICOM sobre Windows, emplear la biblioteca GDCM.

Modificabilidad – Mantenibilidad: para la actualización o adición de nuevas funcionalidades, no es necesario repetir el proceso completo de despliegue, solo se necesita adicionar, desde la interfaz visual “Administrar plugins”, el *plugin* que contiene la funcionalidad o copiarlo al directorio **plugins**, ubicado en la carpeta de instalación.

Reusabilidad - Escalabilidad: las interfaces definidas en el módulo ***Plugins***, pueden emplearse para construir dinámicamente la aplicación, esto significa que la adición de un *plugin* registra automáticamente sus componentes visuales en la aplicación, tales como: desencadenadores, ventanas de configuración y procesadores.

Es necesario destacar que la arquitectura propuesta depende explícitamente del *framework* Qt, por lo que portarla hacia otro *framework* gráfico implica un esfuerzo adicional considerable. Además, la carga dinámica de los plugins de la aplicación aumenta el tiempo de respuesta durante el inicio de la misma.

CONCLUSIONES

Con la realización de este trabajo, se definió y validó una arquitectura de software para el proyecto Vismedic, que permite reducir los problemas de extensibilidad, reusabilidad y dependencias que presenta la arquitectura sobre la que se desarrollan actualmente los productos del proyecto. De esta forma se dio cumplimiento al objetivo propuesto al inicio de la investigación, además:

- La combinación de los estilos arquitectónicos: Arquitectura basada en capas, Arquitectura basada en componentes y Tuberías y filtros, permitió desarrollar una arquitectura que satisface los atributos de calidad reutilización y mantenibilidad.
- El análisis de los productos Voreen, VTK e ITK permitió reutilizar los conceptos fundamentales asociados a una red de flujo de datos (procesadores, puertos, tipos de datos). La incorporación de *plugins* en la creación de la red de flujo de datos constituye el principal aporte de la investigación.
- La configuración dinámica de la red de flujo de datos por parte del usuario, posibilita crear prototipos rápidos de aplicaciones sin necesidad de conocer el código fuente de la aplicación, lo que incrementa la versatilidad de los productos del proyecto.

RECOMENDACIONES

A la arquitectura propuesta en la presente investigación, se le pueden aplicar un conjunto de modificaciones que mejoren su desempeño, por lo que se recomienda:

- Realizar una segunda iteración donde se mitiguen los principales riesgos detectados tras la aplicación del ATAM.
- Evaluar la posibilidad de sustituir la estructura de datos que se emplea para almacenar los procesadores en la red de flujo de datos, por otra estructura que se adapte mejor a la naturaleza de la red, por ejemplo un grafo.

Se recomienda además:

- Valorar la factibilidad de emplear la arquitectura propuesta en otros proyectos relacionados con el procesamiento de imágenes digitales o que empleen de forma general el concepto de redes de flujo de datos.

BIBLIOGRAFÍA

1. *Foundations for the Study of Software Architecture.* Perry, Dewayne E. and Wolf, Alexander L. 4, s.l. : ACM SIGSOFT Software Engineering Notes, Oct 1992, Vol. 17, pp. 40-52.
2. Software Engineering Institute | Carnegie Mellon. [Online] Carnegie Mellon University. [Cited: Diciembre 10, 2012.] http://www.sei.cmu.edu/architecture/start/glossary/community.cfm.
3. ISO/IEC/IEEE 42010 Systems and software engineering - Architecture description. *ISO/IEC/IEEE 42010: Defining architecture.* [Online] [Cited: Enero 10, 2012.] http://www.iso-architecture.org/ieee-1471/defining-architecture.html.
4. *Architectural Blueprints - The "4+1" View Model of Architecture.* Kruchten, Philippe B. 6, s.l. : IEEE SOFTWARE, Nov. 1995, Vol. 12, pp. 42-50.
5. *Microsoft Application Architecture Guide, 2nd Edition.* s.l. : Microsoft, Oct. 2009. ISBN: 9780735627109.
6. Bass, Len, Clements, Paul and Kazman, Rick. *Software Architecture in Practice, Third Edition.* 3. s.l. : Addison-Wesley Professional, Septiembre 2012. p. 4. ISBN: 978-0-321-81573-6.
7. *A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems.* Shaw, Mary and Clements, Paul. Pittsburgh : Carnegie Mellon University, 1997. Proceedings of the 21st International Computer Software and Applications Conference.
8. Fielding, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures.* Irvine : University of California, 2000. Doctoral dissertation.
9. Buschman, Frank, et al. *Pattern-Oriented Software Architecture - A System of Patterns.* Germany : Siemens AG, 1996. Vol. 1. ISBN: 0 471 95869 7.
10. Reynoso, Carlos and Kiccillof, Nicolás. Estilos y Patrones en la Estrategia de Arquitectura de Microsoft. [Online] 2004. [Cited: Noviembre 25, 2012.] www.willydev.net/descargas/prev/Estiloypatron.pdf.
11. Szyperski, Clemens Alden. *Component software: Beyond Object-Oriented programming.* 2nd. s.l. : Addison-Wesley, 2002. ISBN: 0201745720.
12. de la Torre Llorente, César, et al. *Guía de Arquitectura N-Capas orientada al Dominio con .NET 4.0 (Beta).* s.l. : Krasis Press, Marzo 2010. pp. 9-31. ISBN: 978-84-936696-3-8.
13. Cristiá, Maximiliano. *Catálogo Incompleto de Estilos Arquitectónicos.* s.l. : Universidad Nacional de Rosario, 2006.
14. Marquina, Enrnesto. *Guía de Patrones, Prácticas y Arquitectura .NET (Versión 2.0).* [Documento] s.l. : Microsoft Service, Organización Contoso, 2008.

15. Gamma, Erich, et al. *Design Patterns: Elements of Reusable Object-Oriented Software.* s.l. : Addison Wesley, 1995. ISBN: 9781405837309.

16. Larman, Craig. *UML y Patrones: Una introducción al análisis y diseño orientado a objetos y al proceso unificado.* 2nd. s.l. : Prentice-Hall, 2003. ISBN: 8420534382.

17. Geveci, Berk and Schroeder, Will. *The Architecture of Open Source Applications. Elegance, Evolution, and a Few Fearless Hacks.* [ed.] Amy Brown and Greg Wilson. 2011. ISBN: 978-1-257-63801-7.

18. Ibáñez, Luis and King, Brad. *The Architecture of Open Source Applications. Volume II: Structure, Scale, and Few More Fearless Hacks.* [ed.] Amy Brown and Greg Wilson. 2012. Vol. 2. ISBN: 9781105571817.

19. Voreen - Volume Rendering Engine (Official Project Website). *Voreen - Volume Rendering Engine (Official Project Website).* [Online] Visualization & Computer Graphics Research Group, University of Münster, Germany. [Cited: Enero 12, 2013.] http://voreen.uni-muenster.de/.

20. OSGi Alliance. *The OSGi Alliance. OSGi Core Release 5.* March, 2012.

21. Jacobson, Ivar , Booch, Grady and Rumbaugh, James. *El Proceso Unificado de Desarrollo de Software.* [ed.] Andrés Otero. [trans.] Salvador Sánchez, et al. s.l. : Pearson Educacion S.A., 2000. ISBN: 84-7829-036-2.

22. Clements, Paul, Kazman, Rick and Klein, Mark. *Evaluating software architectures: methods and case studies.* s.l. : Addison-Wesley, 2002. ISBN: 9780201704822.

23. Barbacci, Mario, et al. *Quality Atributes (CMU/SEI-95-TR-021).* s.l. : Software Engineering Institute, Carnegie Mellon University, 1995.

24. Booch, Grady, Rumbaugh, James and Jacobson, Ivar. *The Unified Modeling Language User Guide.* s.l. : Pearson Education, 2005. ISBN: 9788131715826.

25. Institute of Electrical and Electronics Engineers. *IEEE Standard Glossary of Software Engineering Terminology - IEEE Std 610.12-1990.* New York : s.n., 1990. ISBN: 1-55937467-X.

26. Pressman, Roger S. Ingeniería del Software. Un enfoque Práctico. 5ta Edición. s.l. : McGraw-Hill. Higher Education, 2002. ISBN: 8448132149.

27. —. Software Engineering. A Practitioner´s Approach. s.l. : Mc Graw Hill. Higher Education, 2010. ISBN: 978-0-07-337597-7.

28. Quality Characteristics for Software Architecture. Losavio, Francisca, et al. Marzo-Abril 2003, Journal of Object Technology, Vols. 2, no.2, pp. 133-150.

29. Bosh, Jan. *Design & Use of Software Architectures. Adopting and evolving a product-line*

approach. s.l. : Addison-Wesley Professional, 2000. ISBN: 978-0201674941.

30. Camacho, Erika, Cardeso, Fabio and Núñez, Gabriel. *Arquitecturas de Software. Guía de Estudio.* Abril 2004.

GLOSARIO DE TÉRMINOS

A

Algoritmo: conjunto ordenado y finito de operaciones que permite obtener la solución de un problema.

D

DICOM: del inglés *Digital Imaging and Communication in Medicine,* es el estándar reconocido mundialmente para el intercambio de imágenes médicas, pensado para el manejo, almacenamiento, impresión y transmisión de las imágenes.

DCMTK: acrónimo del inglés *DICOM Toolkit*, es una colección de bibliotecas y aplicaciones de código abierto que implementa una gran parte del estándar DICOM utilizando los lenguajes ANSI C y C++.

F

Framework: en el desarrollo de software, es una estructura de soporte en la que otro proyecto puede ser organizado y desarrollado. Puede incluir soporte de programas, bibliotecas, un lenguaje interpretado, entre otros software para ayudar a desarrollar y unir los diferentes componentes de un proyecto.

G

GDCM: acrónimo del inglés *Grassroots DICOM* es una biblioteca multiplataforma escrita en C++ para imágenes médicas en formato DICOM.

O

OpenGL: acrónimo del inglés *Open Graphics Library,* es una especificación estándar que define una interfaz de programación de aplicaciones multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D.

P

Plugin: aplicaciones que añaden una funcionalidad adicional o una nueva característica a un software.

Procesamiento digital de imágenes: conjunto de técnicas que se aplican a las imágenes digitales con el objetivo de mejorar la calidad o facilitar la búsqueda de información.

R

Resonancias Magnéticas (RM): La resonancia magnética consiste en la obtención de imágenes radiológicas de la zona anatómica que se desea estudiar mediante el empleo de un campo electromagnético, un emisor/receptor de ondas de radio (escáner) y un ordenador.

Ribbon: interfaz gráfica de usuario, compuesta por una cinta de opciones que contiene las funciones que puede realizar un programa.

T

Tomografías Axiales Computarizadas (TAC): Es un examen médico no invasivo ni doloroso que ayuda al médico a diagnosticar y tratar enfermedades. Las imágenes por TAC utilizan un equipo de rayos X especial para producir múltiples imágenes o visualizaciones del interior del cuerpo, a la vez que utiliza conjuntamente una computadora que permite obtener imágenes transversales del área en estudio. Luego, las imágenes pueden imprimirse o examinarse en un monitor de computadora.