Universidad de las Ciencias Informáticas

Facultad 5

Arquitectura de software para los Laboratorios Virtuales

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

Autor: Alejandro David Merzeau Martínez
Tutor: MSc. Orlay García Ducongé
Co-tutor: Ing. Yaima Fiallo Valle.

"*La Habana, mayo 2012"*

“You can't trust code that you did not totally create yourself. (Especially code from companies that employ people like me)”

Ken Thompson

DECLARACIÓN DE AUTORÍA

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

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

_____

Alejandro D. Merzeau Martínez

_____

Tutor: MSc. Orlay García Ducongé

_____

Co-tutor: Ing. Yaima Fiallo Valle.

DATOS DE CONTACTO

Tutor: MSc. Orlay García Ducongé.
Edad: 27.
Ciudadanía: cubano.
Institución: Universidad de las Ciencias Informáticas (UCI).
Título: MSc. en Informática Aplicada.
Categoría Docente: Instructor.
E-mail: oducunge@uci.cu
Graduado de la Universidad de las Ciencias Informáticas (UCI), con seis años de experiencia en gráficos por computadora y visualización tridimensional.

Co-tutor: Ing Yaima Fiallo Valle.
Edad: 29.
Ciudadanía: cubano.
Institución: Universidad de las Ciencias Informáticas (UCI).
Categoría Docente: Instructor.
E-mail: yfiallo@uci.cu
Graduado de la Universidad de Cienfuegos Carlos Rafael Rodríguez (UCF) y ISPJAE, con siete años de experiencia como ingeniería de software.

AGRADECIMIENTOS

A todas las personas que han colaborado de una forma u otra en la realización de este trabajo y en mi formación tanto personal como profesional.

A mi tutor, que acogió esta tesis como si fuese suya.

A mis grandes amigos Leo, Ede y Manuel.

A todos los amigos de la universidad.

A todos, muchas gracias.

DEDICATORIA

Para mi mamá Yanet, que nunca me ha dicho que no.

Para mi otra mamá Lena, que sabe qué decirme en cada momento.

Para mi papá Alberto, que no necesita decirme mucho.

Para mi hermano Alexander, que lo tengo siempre presente y lo quiero un montón.

Para mi otro hermano Michel, que me enseña cada día que se puede ser mejor.

Para toda mi familia. Para todos mis amigos.

RESUMEN

La continua evolución del mundo del desarrollo de software ha demostrado la importancia de una Arquitectura de Software funcional y estable, que permita agilizar los tiempos de desarrollo y aumentar la reutilización de componentes. El presente trabajo tiene como objetivo definir una propuesta de arquitectura para el desarrollo de laboratorios virtuales. En este se presentan los conceptos de arquitectura y su importancia para el proceso de desarrollo de software. También se realiza un análisis de varios estilos arquitectónicos y patrones de diseño que dieron respuesta al problema planteado.

Basándose en las especificaciones de RUP se describen los requerimientos del sistema, que lograron dar una representación completa de la arquitectura a través de las vistas arquitectónicas propuestas por Kruchten. Igualmente, se detallan las restricciones del diseño e implementación que garantizaron la organización y estandarización del proceso de desarrollo de software. A partir del método ATAM se efectuó la evaluación a la arquitectura propuesta. Asimismo se demostró que la arquitectura propuesta es funcional y candidata a utilizar para el desarrollo de distintos tipos de laboratorios virtuales.

PALABRAS CLAVE

Arquitectura de software, Laboratorio Virtual, Ogre3D, Qt.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

INTRODUCCIÓN

En la actualidad, el constante desarrollo de las Tecnologías de la Información y las Comunicaciones (TIC) ha permitido intervenir en un sin número de procesos. Con mayor o menor grado las TIC han logrado cambiar la manera en que estos procesos ocurren, así como su impacto en la sociedad. Uno de los procesos que se ha visto beneficiado con el avance de las TIC es el docente-educativo, al cual se le han incorporado nuevas herramientas y métodos que tienen como objetivo lograr una mejor formación de los estudiantes.

Los laboratorios virtuales son de estas nuevas tecnologías, que como los tradicionales, tienen como objetivo fundamental poner en práctica los conocimientos obtenidos durante las clases. En estos entornos virtuales no se necesitan de los materiales y herramientas reales para la realización de las prácticas, por lo que constituyen una opción viable ante la falta de recursos u otros impedimentos existentes.

En la Universidad de las Ciencias Informáticas, específicamente en el Centro de Informática Industrial (CEDIN) se encuentra el proyecto “Laboratorios Virtuales”, que tiene como objetivo desarrollar aplicaciones de apoyo a la educación. Este tiene como experiencia el desarrollo de tres de estas aplicaciones. Después de la entrega de estos productos y su satisfactoria aceptación por parte del cliente, se contrataron nuevos productos. Bajo el contexto del desarrollo previo de los productos entregados y los contratiempos encontrados en todo el ciclo de desarrollo de software se puede afirmar que; la arquitectura de software existente no cumple con las necesidades del proyecto para iniciar el desarrollo de estos nuevos productos. Dentro de los principales problemas se puede mencionar la inexistencia de una estructura lógica, lo que dificulta el entendimiento de esta por parte del equipo de desarrollo. No hay documentación asociada a la arquitectura, lo que obstaculiza el proceso de su aprendizaje para desarrolladores que no estaban involucrados con el proyecto. Los elementos en la arquitectura de software están altamente acoplados y son dependientes entre sí. Esto significa que la realización de nuevos requisitos puede tornase una tarea compleja, aún más si los requisitos no guardan relación alguna con los existentes. Dada la falta de estándares, es

notable la desorganización en aspectos como la presentación de datos al usuario y el código fuente.

La presencia de estos problemas hace de la arquitectura de software existente una propuesta poco factible a utilizar. El coste asociado a su actualización podría ser alto, teniendo en cuenta el poco nivel de reutilización que se puede obtener. Además, se necesita incorporar nuevas características que mejoren y agilicen el proceso de desarrollo y dar la posibilidad de brindar soporte de manera sencilla.

Dada la situación anteriormente expuesta se define como **problema científico**: ¿Cómo establecer la comunicación entre los distintos componentes de un Laboratorio Virtual, como base para un desarrollo estandarizado?

Como **objetivo** del trabajo se propone desarrollar una arquitectura basada en componentes con un alto grado de cohesión y bajo acoplamiento para el desarrollo de distintos tipos de laboratorios virtuales.

Como resultado del problema de investigación, se define como **objeto de estudio** las arquitecturas de software, enfocándose en el **campo de acción** las arquitecturas de software para laboratorios virtuales.

Para darle cumplimiento al objetivo planteado se proponen las siguientes **tareas investigativas:**

1. Búsqueda bibliográfica sobre el tema arquitecturas de software con el objetivo de elaborar un marco teórico.
2. Análisis de la arquitectura anterior para identificar elementos de interés.
3. Identificación y descripción de los requerimientos del sistema.
4. Selección de las herramientas para darle solución al problema.
5. Diseño de la solución.
6. Implementación de una arquitectura para laboratorios virtuales.
7. Validación la propuesta arquitectónica.

Para todo el proceso de investigación y elaboración de este trabajo se utilizarán varios **métodos científicos de investigación** como:

Histórico – Lógico: Método teórico que se utiliza para analizar la evolución y las tendencias actuales de las arquitecturas de software y los laboratorios virtuales.

Analítico – Sintético: Método teórico que se utiliza para extraer y analizar la información sobre las principales arquitecturas de software asociadas a entornos virtuales.

Inductivo – Deductivo: Este método se utiliza para darle solución al problema planteado a través de conocimientos generales sobre el tema de investigación, llegando así a conclusiones particulares.

Consultas de fuente de información: Método empírico que se utiliza para la consulta de fuentes bibliográficas durante la investigación.

Observación: Método empírico que se utiliza para observar los resultados obtenidos en la caracterización e identificación de los principales tipos de arquitecturas de software y decidir luego cuál o cuáles serán más adecuadas.

El presente trabajo está estructurado en capítulos de la siguiente manera:

Capítulo 1: Fundamentación Teórica.

Este capítulo contiene las principales definiciones relacionadas con las arquitecturas de software. Además se encuentra una síntesis de los estilos arquitectónicos y patrones de diseño que se utilizaron en la propuesta de arquitectura. Se describen también los principales conceptos asociados a los laboratorios virtuales.

Capítulo 2: Solución Propuesta.

Este capítulo contiene el diseño de la solución, haciendo énfasis en las diferentes características del sistema y sus ventajas como arquitectura de software para el proyecto de Laboratorios Virtuales.

Capítulo 3: Evaluación de la arquitectura.

Este capítulo expone los diferentes conceptos asociados a la evaluación de la arquitectura. También se efectúa la evaluación de la arquitectura haciendo uso del método ATAM.

CAPÍTULO 1: FUNDAMENTACIÓN TEÓRICA

1.1 Arquitectura de Software

El término Arquitectura de Software (AS) es abordado por un gran número de autores. Aunque todos aportan sus puntos de vistas en función de sus propósitos, en general existen importantes puntos de contactos que nos facilita la correcta interpretación del término.

Philippe Kruchten[1] refiriéndose a la AS expresa:

"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." (1)

Paul Clements, la define como:

"La arquitectura de software es, a grandes rasgos, una vista del sistema que incluye los componentes principales del mismo, la conducta de esos componentes según se le percibe desde el resto del sistema y las formas en que los componentes interactúan y se coordinan para alcanzar la misión del sistema. La vista arquitectónica es una vista abstracta, aportando el más alto nivel de comprensión y la supresión o diferimiento del detalle inherente a la mayor parte de las abstracciones." (2)

La IEEE propone: "La Arquitectura del Software es la organización fundamental de un sistema formada por sus componentes, las relaciones entre ellos y el contexto en el que se implantarán, y los principios que orientan su diseño y evolución." (3)

En el libro "El Proceso Unificado de Software" los autores refieren que:

"La arquitectura de software se centra en los elementos estructurales significativos del sistema, como subsistemas, clases, componentes, nodos y las colaboraciones que tienen lugar entre estos elementos a través de las interfaces. Los casos de uso dirigen la arquitectura para hacer que el sistema proporcione la funcionalidad y uso deseado, alcanzando a la vez objetivos de rendimientos razonables. Una arquitectura debe ser

[1] Director de Rational Unified Process (RUP) y desarrollador del modelo 4+1.

completa, pero también debe ser suficientemente flexible como para incorporar nuevas funciones, y debe soportar la reutilización del *software* existente." (4)

Len Bass[2] la describe como:

"…la estructura de las estructuras del sistema, la cual comprende los elementos del software, las propiedades de esos elementos visibles externamente, y las relaciones entre ellos." (5)

Dentro de los puntos de interés que identifican estos conceptos podemos constatar que los autores se refieren a la AS como un elemento guía para desarrollar el software, una aproximación inicial a la solución del problema.

Por esta razón la AS constituye los pilares fundamentales donde se apoyará todo el proceso de desarrollo de un *software*. En la actualidad se ha profundizado más sobre el tema de las AS así como su importancia para el desarrollo de un producto de *software* de buena calidad. Un sistema de *software* requiere una arquitectura para que los desarrolladores puedan avanzar hasta lograr una visión común, más en nuestros días cuando los sistemas son cada vez más complejos.

Los problemas de diseño van más allá de los algoritmos y estructuras de datos de la computación; el diseño y especificación de la estructura global del sistema es un nuevo tipo de problema (6).

Además, generalmente existen sistemas que realizan algunas de las funciones que se requieren desarrollar. Saber identificar qué hace y qué elementos del código pueden reutilizarse, añade complejidad al desarrollo (4).

1.2 Necesidad de una arquitectura.

Entre las principales causas por lo cual es necesaria una arquitectura se pueden citar:

- Comprender el sistema: Para que un proyecto desarrolle un sistema, dicho sistema debe ser comprendido por todos los que vayan a intervenir en él. Lograr que los sistemas modernos sean comprensibles es un reto importante por muchas razones:

[2] Autor de más de una docena de artículos de ingeniería de software y miembro del Instituto de Ingeniería de Software en Estados Unidos.

  - Abarcan un comportamiento complejo.
  - Operan en entornos complejos.
  - Son tecnológicamente complejos.
  - Deben satisfacer demandas individuales de la organización.
  - A menudo, combinan computación distribuida, productos y plataformas comerciales y reutilizan componentes y *frameworks* de trabajo (4).

- Organizar el desarrollo: Cuanto mayor sea la organización del proyecto de *software*, mayor será la sobrecarga de comunicación entre los desarrolladores para intentar coordinar sus esfuerzos. Dividiendo el sistema en subsistemas, con las interfaces claramente definidas y con un responsable o un grupo de responsables establecido para cada subsistema, el arquitecto puede reducir la carga de comunicación entre los grupos de trabajo de los diferentes subsistemas (4).
- Fomentar la reutilización: Mientras mayor cantidad de componentes reutilizables existan en el sistema, menor serán los tiempos de desarrollo y costo de otros sistemas futuros (4).
- Hacer evolucionar al sistema: El sistema debe ser en sí mismo flexible a los cambios o tolerante a los cambios, debe ser capaz de evolucionar sin problemas puesto que las arquitecturas de los sistemas pobres suelen degradarse con el paso del tiempo y necesitan ser parcheadas hasta que al final no es posible actualizarla con un coste razonable (4).

La continua evolución del mundo del desarrollo de *software* ha demostrado la importancia de una AS funcional y estable, así se señala en el libro, *"Software Architecture: Perspectives on an emerging discipline"*, donde se explica que:

"El disponer de componentes de software no es suficiente para desarrollar aplicaciones, ya provengan estos de un mercado global o sean desarrollados a medida para la aplicación. Un aspecto crítico a la hora de construir sistemas complejos es el diseño de la estructura del sistema, y por ello el estudio de la arquitectura de software se ha convertido en una disciplina de especial relevancia en la ingeniería del software." (7)

Barry Boehm[3] expone:

"Si un proyecto no ha logrado una arquitectura del sistema, incluyendo su justificación, el proyecto no debe empezar el desarrollo en gran escala. Si se especifica la arquitectura como un elemento a entregar, se la puede usar a lo largo de los procesos de desarrollo y mantenimiento." (8)

A pesar de la variedad de definiciones y puntos de vistas diferentes, ningún autor niega la importancia que tiene la AS para el desarrollo de *software* con calidad. Así lo demuestran las definiciones ya vistas de Kruchten, Jacobson y Clements, este último añadiéndole cierta responsabilidad a los otros elementos que garantizan la buena calidad del software cuando expresa:

"La arquitectura tiene influencia en la calidad pero no la garantiza." (5)

1.3 Áreas de investigación de las Arquitecturas de Software

A medida que el estudio de las AS logró posicionarse como un elemento de interés en la disciplina de ingeniería de *software*, los autores han tratado de delinear los campos o temas principales de la AS, donde se destacan:

- Lenguajes de descripción de arquitecturas.
- Fundamentos formales de la AS (bases matemáticas, caracterizaciones formales de propiedades extra-funcionales tales como mantenibilidad, teorías de la interconexión, etcétera).
- Técnicas de análisis arquitectónicas.
- Métodos de desarrollo basados en arquitectura.
- Recuperación y reutilización de arquitectura.
- Codificación y guía arquitectónica.
- Herramientas y ambientes de diseño arquitectónico.
- Estudio de casos (9).

También Clements propone cinco temas fundamentales en torno de los cuales se agrupa la disciplina: (10)

[3] Especialista en gestión de riesgo y conocido creador del *Constructive Cost Model* (COCOMO) y del método de desarrollo en espiral.

- Diseño o selección de la arquitectura: ¿Cómo crear o seleccionar una arquitectura en base de requerimientos funcionales, de rendimiento o de calidad?
- Representación de la arquitectura: ¿Cómo comunicar una arquitectura? Este problema se ha manifestado como el problema de la representación de arquitecturas utilizando recursos lingüísticos, pero el problema también incluye la selección del conjunto de información a ser comunicada.
- Evaluación y análisis de la arquitectura: ¿Cómo analizar una arquitectura para predecir cualidades del sistema en que se manifiesta?
- Desarrollo y evolución basados en arquitectura: ¿Cómo construir y mantener un sistema dada una representación que se cree, es la arquitectura que resolverá el problema correspondiente?
- Recuperación de la arquitectura: ¿Cómo hacer que un sistema evolucione cuando los cambios afectan su estructura? Para los sistemas de los que se carezca de documentación confiable, esto involucra primero una "arqueología arquitectónica" que extraiga su arquitectura.

El estudio de la AS todavía es muy joven, los campos más promisorios tienen que ver con el tratamiento sistemático de estilos, el desarrollo de lenguajes de descripción arquitectónica, la formulación de metodologías y el trabajo con patrones de diseño. Las AS se encuentran en su fase de desarrollo y extensión, pero las ideas y herramientas distan aún de estar maduras (11).

1.4 Estilos Arquitectónicos

Los estilos arquitectónicos al igual que su homónimo en la arquitectura tradicional representan diferentes tendencias que de alguna manera u otra han sido reconocidas y utilizada por los especialistas. Un estilo describe una clase de arquitectura, o piezas identificables de las arquitecturas empíricamente dadas. Esas piezas se encuentran repetidamente en la práctica, trasuntando la existencia de decisiones estructurales coherentes. Una vez que se han identificado los estilos, es lógico y natural pensar en re-utilizarlos en situaciones semejantes que se presenten en el futuro (12).

Después de recibir nombres variados tales como: "clases de arquitectura", "tipos arquitectónicos", "arquetipos recurrentes", "especies", "paradigmas topológicos" entre

otras cualificaciones desde hace ya un tiempo se ha instituido la definición de “estilos” o alternativamente “patrones de arquitectura” aunque esta última tiende a confundir con “patrones de diseño”. Llamarlas estilos subraya explícitamente la existencia de una taxonomía de las alternativas estilísticas; llamarlas patrones, por el contrario, establece su vinculación con otros patrones posibles pero de distinta clase: de diseño, de organización o proceso, de código, de interfaz de usuario, de prueba, de análisis (9).

El proceso de definición de *“*estilo arquitectónico*”* comenzó con el mismo surgimiento del término AS, entre las primeras definiciones se encuentra la propuesta de Perry y Wolf que expresan:

“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.” (13)

Shaw y Clements definen los estilos arquitectónicos como un conjunto de reglas de diseño que identifican las clases de componentes y conectores que se pueden utilizar para componer en sistema o subsistema, junto con las restricciones locales o globales de la forma en que la composición se lleva a cabo (14). Klein y Kazman proponen una definición según la cual un estilo arquitectónico es una descripción del patrón de los datos y la interacción de control entre los componentes, ligada a una descripción informal de los beneficios e inconvenientes aparejados por el uso del estilo. Los estilos arquitectónicos, afirman, son artefactos de ingeniería importantes porque definen clases de diseño junto con las propiedades conocidas, asociadas a ellos. Ofrecen evidencia basada en la experiencia sobre la forma en que se ha utilizado históricamente cada clase, junto con un razonamiento cualitativo para explicar por qué cada clase tiene esas propiedades específicas (15).

Robert Allen y David Garlan asimilan los estilos arquitectónicos a descripciones informales de arquitecturas basadas en una colección de componentes computacionales, junto a una colección de conectores que describen las interacciones entre los componentes (16).

También es válida la definición de estilo arquitectónico como un conjunto coordinado de restricciones arquitectónicas que establece los roles y rasgos de los elementos y las relaciones permitidas entre ellos (17).

Se puede apreciar que todas estas definiciones tratan a los estilos como una entidad que ocurre a un nivel sumamente abstracto, sugiriendo una estructura genérica para la organización de componentes de ciertas familias de sistemas, independientemente del contexto en que estas se desarrollen.

El conjunto total de estilos arquitectónicos existentes en la actualidad no supera la decena, pero una formalización del nombre y categoría de los mismos pareciera imposible en la actualidad:

“Mientras que en los inicios de la arquitectura de software se alentaba la idea de que todas las estructuras posibles en el diseño de software serían susceptibles de reducirse a una media docena de estilos básicos, lo que en realidad sucedió fue que en los comienzos del siglo XXI se alcanzó una fase barroca y las enumeraciones de estilos se tornaron más detalladas y exhaustivas. Cuando las clasificaciones de estilos se tornan copiosas, daría la impresión que algunos sub-estilos se introducen en el cuadro por ser combinatoriamente posibles y no tanto porque existan importantes implementaciones de referencia o porque sea técnicamente necesario.” (9)

Teniendo en cuenta la descripción de Reynoso (9), una posible categorización de los estilos arquitectónicos es:

- **Estilos de Flujo de Datos**

  Tubería y filtros.

- **Estilos Centrados en Datos**

  Arquitecturas de Pizarra o Repositorio.

- **Estilos de Código Móvil**

  Arquitectura de Máquinas Virtuales.

- **Estilos *Peer-to-Peer***

  Arquitecturas Basadas en Eventos.

  Arquitecturas Orientadas a Servicios (SOA).

  Arquitecturas Basadas en Recursos.

- **Estilos de Llamada y Retorno**

  Model-View-Controller.

Arquitecturas en Capas.

Arquitecturas Orientadas a Objetos.

Arquitecturas Basadas en Componentes.

- **Estilos Derivados**

  C2.

  GenVoca.

  Rest.

- **Estilos Heterogéneos**

  Sistema de Control de Procesos.

  Arquitecturas Basadas en Atributos.

Se considera que la importancia fundamental de un estilo radica en proveer al desarrollador, o equipo de desarrollo, de una forma de identificarse rápidamente con el diseño. Esto permite que se puedan esclarecer algunas dudas respecto al diseño con solo realizar asociaciones entre este y el estilo arquitectónico propuesto.

1.4.1 Estilo basado en Capas.

El estilo basado en capas funciona como 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 (6).

En la práctica, las capas suelen ser entidades complejas, compuestas de varios paquetes o subsistemas.

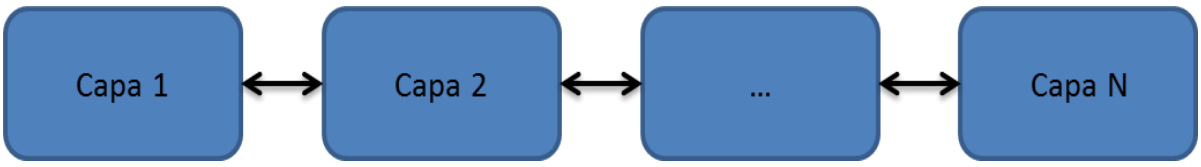

Figura 1. Estilo basado en capas.

Ventajas: El estilo soporta un diseño basado en niveles de abstracción crecientes, lo cual a su vez permite a los implementadores la partición de un problema complejo en una

secuencia de pasos incrementales. Admite muy naturalmente optimizaciones y refinamientos, proporcionando un amplio nivel de reutilización. Al igual que los tipos de datos abstractos, se pueden utilizar diferentes implementaciones o versiones de una misma capa en la medida que soporten las mismas interfaces de cara a las capas adyacentes. Esto conduce a la posibilidad de definir interfaces de capa estándar, a partir de las cuales se pueden construir extensiones o prestaciones específicas (9).

Desventajas: Muchos problemas no admiten un buen mapeo en una estructura jerárquica. Incluso cuando un sistema se puede establecer lógicamente en capas, consideraciones de rendimiento pueden requerir acoplamientos específicos entre capas de alto y bajo nivel. Además los cambios en las capas de bajo nivel tienden a filtrarse hacia las de alto nivel, también se admite que la arquitectura en capas ayuda a controlar y encapsular aplicaciones complejas, pero complica no siempre razonablemente las aplicaciones simples (18).

1.4.2 Estilo basado en Componentes

Una arquitectura basada en componentes describe una aproximación de ingeniería de software al diseño y desarrollo de un sistema. Esta arquitectura se enfoca en la descomposición del diseño en componentes funcionales o lógicos que expongan interfaces de comunicación bien definidas. Esto provee un nivel de abstracción mayor que los principios de orientación por objetos y no se enfoca en asuntos específicos de los objetos como los protocolos de comunicación y la forma como se comparte el estado (19).

Un componente es una unidad de composición de aplicaciones, que posee un conjunto de interfaces especificadas contractualmente y dependencias del contexto explícitas. Que ha de poder ser desarrollado, adquirido, incorporado al sistema y compuesto con otros componentes de forma independiente, en tiempo y espacio (20).

El estilo de arquitectura basado en componentes tiene como principales características:

- Ser un estilo de diseño para aplicaciones compuestas de componentes individuales.

- Pone énfasis en la descomposición del sistema en componentes lógicos o funcionales que tienen interfaces bien definidas.

- Define una aproximación de diseño que usa componentes discretos, los que se comunican a través de interfaces que contienen métodos, eventos y propiedades (19).

Ventajas:

Facilidad de Instalación: Cuando una nueva versión esté disponible, se podrá remplazar la versión existente sin impacto en otros componentes o el sistema como un todo.

Costos reducidos: El uso de componentes de terceros permite distribuir el costo del desarrollo y del mantenimiento.

Facilidad de desarrollo: Los componentes implementan una interface bien definida para proveer la funcionalidad permitiendo el desarrollo sin impactar otras partes del sistema.

Reusable: El uso de componentes reutilizables significa que ellos pueden ser usados para distribuir el desarrollo y el mantenimiento entre múltiples aplicaciones y sistemas.

Facilidad de prueba: El uso de compontes en la aplicación permite probarlos a cada uno como una unidad independiente facilitando más tarde las tareas de pruebas (21).

Desventajas:

Evolución de los componentes: La gestión de la evolución es un problema serio, pues en los sistemas grandes han de poder coexistir varias versiones de un mismo componente.

Particularización: Cómo particularizar los servicios que ofrece un componente para adaptarlo a las necesidades y requisitos concretos de nuestra aplicación, sin poder manipular su implementación (21).

Con una arquitectura basada en componentes no se plantea el desarrollo de una aplicación a la medida, sino el desarrollo de un sistema compuesto por elementos que permiten su reutilización.

1.5 Patrones de Arquitectura

Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno y describe también el núcleo de su solución, de forma que puede utilizarse un millón de veces sin hacer dos veces lo mismo (22).

Los patrones de arquitectura expresan esquemas de organización estructural fundamentales para los sistemas de software. Proporcionan un conjunto de subsistemas predefinidos, especifican sus responsabilidades e incluyen guías y lineamientos para organizar las relaciones entre ellos (23).

1.6 Estilos y Patrones de Arquitectura

Es perceptible que en las definiciones de estilos arquitectónicos y patrones arquitectónicos existen rasgos comunes. Incluso los nombres de algunos estilos y patrones arquitectónicos coinciden en muchos textos, este fenómeno es abordado por Reynoso quien lo describe de la siguiente manera.

“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. Algunas formulaciones que describen patrones pueden leerse como si se refirieran a estilos, y también viceversa. 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”. (9)

Lo cierto es que más allá de los aspectos académicos y estilísticos de cómo tratar estos términos, su mayor importancia radica en saber aplicar en una situación determinada cada estilo, patrón arquitectónico o cualquier otra clasificación que se le quiera dar. Es por ello que la mayoría de los textos consultados traten de ofrecer un punto de vista más orientado hacia su aplicación en la industria así como los beneficios y conflictos que conlleva su uso.

1.7 Patrones de Diseño.

Cuando se menciona el tema de patrones de diseño resulta necesario tomar en cuenta el libro *"Design Patterns: Elements of Reusable Object Oriented Software"*. Este influyente material fue escrito por Erich Gamma, Richard Helm, Ralph Johnson y John Vlisides, también conocidos como "*Gang of Four"* (GoF). En él se presenta un catálogo de 23 patrones divididos en 3 categorías: creacionales, estructurales y comportamiento. Los autores presentan los patrones de diseño a partir de sistemas reales. Es por esta razón que estos patrones no son elementos puramente teóricos, sino que están fundados sobre una fuerte base práctica.

Un patrón de diseño es una descripción de clases y objetos que se comunican entre sí, adaptados para resolver un problema general de diseño en un contexto particular (24).

En la siguiente sección se describen algunos de los patrones de diseño que resultan de interés para el desarrollo de la AS.

Patrón de diseño *Chain of Responsibility*

Evita acoplar el emisor de una petición a su receptor, al dar a más de un objeto la posibilidad de responder a la petición. Crea una cadena con los objetos receptores y pasa la petición a través de la cadena hasta que esta sea tratada por algún objeto (24).

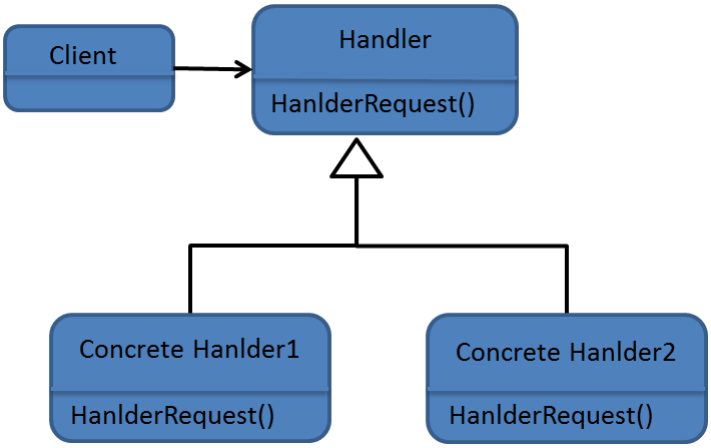

Figura 2. Patrón de diseño *Chain of Responsibility*.

Patrón de diseño *Singleton*

Garantiza que una clase sólo tenga una instancia, y proporciona un punto de acceso global a ella (24).

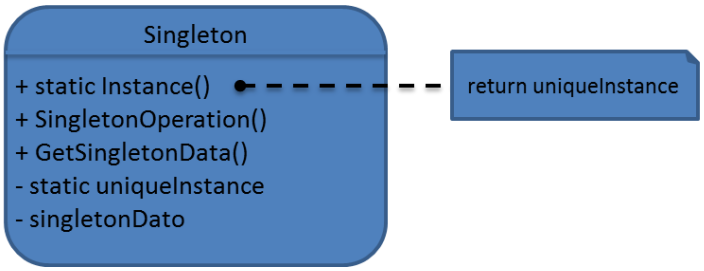

Figura 3. Patrón de diseño *Singleton.*

Patrón de diseño *Adapter*

Convierte la interfaz de una clase en otra distinta, que es la que esperan los clientes. Permiten que cooperen clases que de otra manera no podrían por tener interfaces incompatibles (24).

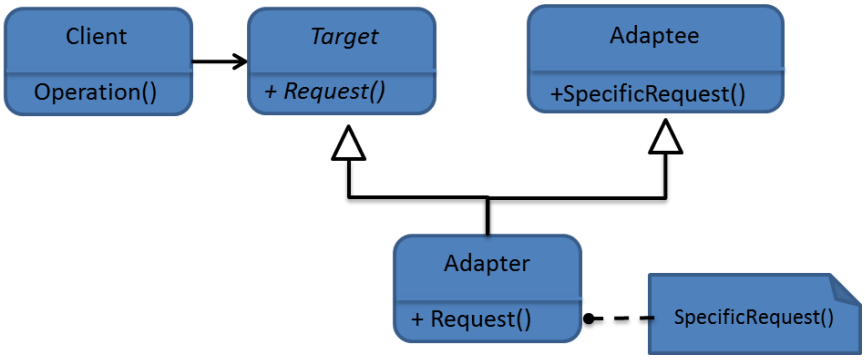

Figura 4. Patrón de diseño *Adpater.*

Patrón de diseño *Factory Method*

Define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan qué clase instanciar. Permite que una clase delegue en sus subclases la creación de objetos (24).

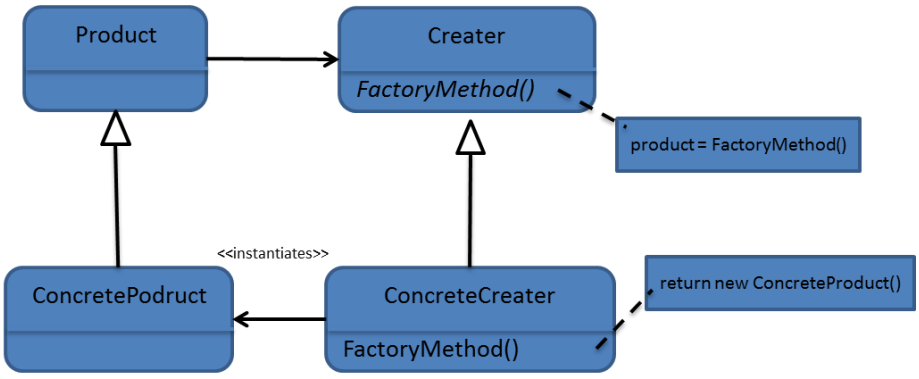

Figura 5. Patrón de diseño *Factory Method*.

En la práctica los patrones de diseño son usados con alguna que otra variación teniendo en cuenta el contexto específico a desarrollar. Incluso muchas veces estos están relacionados entre sí. Por ejemplo: es bastante común encontrar los patrones creacionales junto con el patrón *Singleton*. Y es que en algunas situaciones conviene tener centralizado la administración de determinados tipos de entidades para evitar redundancias innecesarias.

El uso de los patrones de diseño no es de carácter obligatorio y por lo tanto no debe ser forzado. De hecho, los miembros del mentado GoF destacan que los patrones no son un destino sino un punto de partida.

De forma general con el uso de estos patrones se propone:

- Evitar la reiteración en la búsqueda de soluciones a problemas ya conocidos y solucionados anteriormente.
- Formalizar un vocabulario común entre desarrolladores.
- Estandarizar el modo en que se realiza el diseño.

Otros elementos reconocidos como patrones, aunque no específicamente de diseño, son los GRASP (*General Responsibility Assignment Software Patterns*), que describen los principios fundamentales de diseño de objetos para la asignación de responsabilidades. Constituyen un apoyo para la enseñanza que ayuda a entender el diseño de objeto

esencial y aplica el razonamiento para el diseño de una forma sistemática, racional y explicable.

Algunos Patrones GRASP son*:* (25)

- Experto.
- Creador.
- Alta cohesión.
- Bajo acoplamiento.
- Controlador.

1.8 Laboratorios Virtuales

El término Laboratorio Virtual está estrechamente vinculado al desarrollo de las TIC. Estos son producto del continuo avance tecnológico y el desarrollo de alternativas para mejorar la educación y enseñanza en los estudiantes.

Se define Laboratorio Virtual como:

"Espacio electrónico de trabajo concebido para la colaboración y la experimentación a distancia con objeto de investigar o realizar otras actividades creativas, y elaborar y difundir resultados mediante tecnologías difundidas de información y comunicación." (26)

Otra definición con un punto de vista más amplio es:

"Una simulación en computadora de una amplia variedad de situaciones, desde prácticas manipulables hasta visitas guiadas." (27)

Un laboratorio virtual es diferente de un "laboratorio verdadero" o de un "laboratorio tradicional". Sin embargo, no se considera que el laboratorio virtual vaya a suplantar a los verdaderos laboratorios o competir con ellos. En cambio, los laboratorios virtuales constituyen una posible extensión de los verdaderos laboratorios y abren nuevas perspectivas que no se podían explorar completamente, dentro de un laboratorio verdadero, a un costo asequible (26).

1.8.1 Tipos de Laboratorios Virtuales

Para el estudio y desarrollo de los laboratorios virtuales se han dividido en tres grupos fundamentales, teniendo en cuenta sus características particulares:

- Laboratorios virtuales *desktop*: Están desarrollados como un programa independiente para ser ejecutados en los ordenadores, no requieren de un servidor web.
- Laboratorios virtuales web: Este tipo de laboratorios se basa en un software que depende de los recursos de un servidor determinado.
- Laboratorios remotos: Estos laboratorios requieren de equipos servidores específicos que les den acceso a las máquinas a operar de forma remota, y no pueden ofrecer su funcionalidad ejecutándose de forma local.

1.8.2 Ventajas de los Laboratorios Virtuales

Según Calvo (28), algunas de las ventajas asociadas al uso de los laboratorios virtuales son:

- Familiarizarse con el experimento: evitando que los estudiantes puedan acudir al aula sin haber realizado trabajo previo.
- Disminución del uso incorrecto del equipamiento: frecuentemente los dispositivos utilizados en laboratorios reales son delicados, lo que se acentúa si se les hace trabajar fuera de las condiciones de trabajo para las que están diseñados.
- Formar en metodologías de trabajo: en su futura vida laboral los estudiantes habitualmente construirán primero modelos matemáticos de los sistemas que simularán bajo diferentes circunstancias como paso previo a construir prototipos, mucho más caros, con los que experimentar.
- Manejo de herramientas informáticas actuales: en la vida profesional, e incluso en la vida diaria, la destreza en el uso de las herramientas informáticas, es un elemento diferenciador. Con ello se consigue aportar al alumno una serie de conocimientos transversales que sin ser el objetivo principal del laboratorio que se esté diseñando, le servirán en muchos ámbitos del futuro.
- Repetitividad de los experimentos: dado que el comportamiento de los sistemas a estudiar se obtiene mediante el modelado matemático de la realidad, los alumnos pueden repetir de forma totalmente fidedigna las condiciones bajo las que se realizaron los experimentos y reproducirlos en caso de necesidad, con la seguridad de que el resultado será el mismo que ellos vieron en su momento.

1.8.3 Ejemplos de Laboratorios Virtuales

Se pueden localizar laboratorios virtuales de diferentes tipos y materias. Entre los ejemplos webs se citan:

LiveChem de la universidad Oxford, es un laboratorio virtual web para el aprendizaje de la química. Entre sus contenidos se puede destacar el estudio de los iones en disolución, mezcla de sustancias químicas, simetría molecular entre otros (29).

AutomatL@bs es una red de laboratorios virtuales para la enseñanza de la automática que se constituye mediante la integración de los recursos que aportan los grupos que participan en el proyecto. Proporciona un sistema de reserva de tiempos para la realización de los experimentos y un entorno de trabajo común que facilita su aprendizaje por parte del alumno (30).

Algunos de laboratorios remotos reconocidos son:

Automatic Control Telelab, laboratorio remoto con experimentos remotos sobre diferentes plantas (motores, levitadores magnéticos, helicópteros, robots móviles, tanques de agua), y que permite introducir código de control desarrollado por el usuario. Contempla incluso el desarrollo de prácticas de identificación de sistemas (31).

Laboratorio Remoto de Automática Industrial: laboratorio remoto con acceso a equipos industriales, como una planta piloto para la realización de experiencias de control de operación y supervisión remota, maquetas de procesos de control sobre variables de nivel, caudal, temperatura y otros equipos de automatización (32).

Entre los laboratorios virtuales desktop que podemos encontrar:

VLabQ simulador interactivo para prácticas de laboratorio de química, creado por *Sibees Soft* que utiliza equipos y procedimientos estándares para simular los procesos que intervienen en un experimento o práctica. Contiene los instrumentos necesarios al igual que un laboratorio real, tales como: vasos de precipitados, matraces Erlenmeyer, matraz

de balón, reactor, buretas, probetas, pipetas, tubo de ensayo, termómetros, balanzas entre otros (33).

Interactive Physics permite modelar, simular y explorar gran variedad de fenómenos físicos controlando una gran cantidad de variables como gravedad, fuerza, velocidad, masa y posición (34).

1.9 Análisis de la Arquitectura de Software previa

La arquitectura previa fue creada con el objetivo de desarrollar laboratorios virtuales en tres dimensiones. Esta arquitectura cumplió con los objetivos planteados al inicio del desarrollo del sistema aunque muchas veces la propia estructura monolítica obstaculizó enormemente el proceso de desarrollo y originó la aplicación de parches al sistema. Entre las principales dificultades encontradas podemos mencionar:

- Portabilidad: presentaba problemas de compatibilidad entre los sistemas operativos Linux y Windows.
- Configurabilidad: no permitía configurar las propiedades del sistema en dependencia al *hardware* existente.
- Integridad: no existía una arquitectura de la información estandarizada, lo que implicaba que los mensajes fueran distintos en cuanto a formato y contenido.
- Modificabilidad: no permitía actualización de los compontes de forma sencilla.
- Reusabilidad y escalabilidad: elementos altamente acoplados y dependientes por lo que dificultaba la posible reutilización de estos y su extensión a otros tipos de laboratorios.
- Mantenibilidad: no existía una lógica clara y funcional, lo que implicaba dificultades en el desarrollo y soporte posterior a la aplicación.

1.10 Consideraciones parciales.

El estudio realizado ha permitido evaluar la importancia que tiene la arquitectura de software para el desarrollo de laboratorios virtuales o cualquier otro sistema informático. Igualmente, se analizaron los estilos que ayudarán a formar una visión común de la arquitectura por parte de los desarrolladores. En la búsqueda de una estrategia a seguir

para lograr una buena implementación se revisaron diferentes patrones de diseño. Estos sin ser elementos que tienen que estar presentes de manera obligatoria en el diseño si representan un buen punto de partida para llegar a soluciones de problemas muy comunes.

Se pudo apreciar la presencia de una gran cantidad de laboratorios virtuales en el mercado. Sin embargo muy pocos presentan contenidos interactivos utilizando imágenes 3d. Característica que debe ser explotada por los laboratorios virtuales a realizar para lograr mayor aceptación por parte de los clientes.

CAPÍTULO 2: SOLUCIÓN PROPUESTA

En este capítulo se describe en detalles la AS mediante las vistas arquitectónicas y las restricciones del diseño e implementación. También se especifican las herramientas que se utilizarán en el desarrollo del sistema así como se recogen diferentes elementos que proporcionan una base para el entendimiento de la AS propuesta.

Teniendo en cuenta el estudio realizado en el capítulo anterior se asumen las siguientes consideraciones para la realización de este trabajo:

La AS propuesta tiene como objetivo ser la base fundamental para el desarrollo de los laboratorios virtuales, es por ello que se puntualiza “laboratorio virtual” como:

Sistema computacional que pretende simular el ambiente de un laboratorio real; en el cual se visualizan instrumentos y fenómenos a través de imágenes y animaciones, con el propósito de tratar de recrear la realidad, la cotidianeidad y la interactividad que se logra en un laboratorio tradicional. Estas impresiones y emociones, aunque sean ajenas al proceso de fijación del conocimiento, forman parte importante de la experiencia adquirida por el estudiante y dicha experiencia juega un papel fundamental en el desarrollo y el entendimiento del conocimiento.

Asimismo se define AS como:

La Arquitectura de Software es la organización fundamental de un sistema formada por sus componentes, las relaciones entre ellos y el contexto en el que se implantarán. Además constituye la plataforma principal para satisfacer la mayor funcionalidad y los requerimientos de desempeño de un sistema.

Otras definiciones que se proponen son:

Componente: Unidad de composición de aplicaciones, que posee un conjunto de interfaces especificadas contractualmente y dependencias del contexto explícitas.

Componente estático: *Componente que se distribuye en forma de biblioteca dinámica o estática. Es necesario hacer un enlace estático en tiempo de compilación para su uso.*

Componente dinámico: *Componente que se distribuye en forma de biblioteca dinámica. No es necesario hacer un enlace estático para su uso.*

2.1 Modelo de Dominio

Como parte del proceso de conceptualización del problema se realiza un modelo de domino que contribuye a comprender mejor la estructura y los conceptos asociados al funcionamiento del sistema.

El modelo de domino es un diagrama de UML, específicamente un diagrama de clases, que permite mostrar los principales conceptos, y relaciones, que intervienen en el dominio del problema. Un modelo de dominio captura los tipos más importantes de objetos que existen o los eventos que suceden en el entorno donde estará el sistema. A continuación se muestra el diagrama de clases del modelo de dominio:

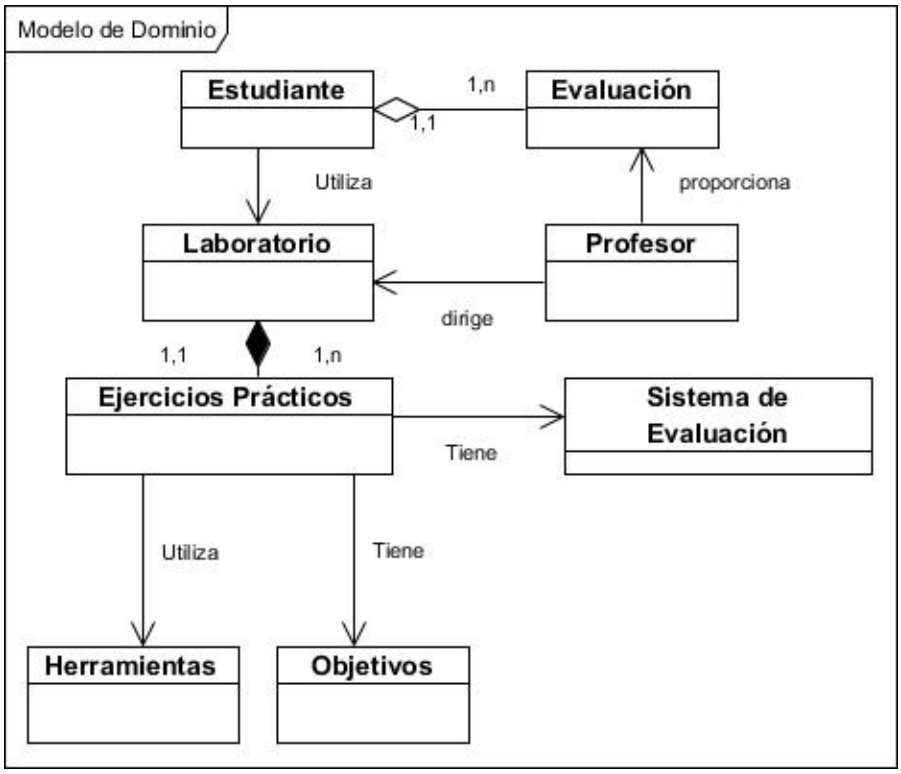

Figura 6. Modelo de dominio.

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

Estudiante: Esta clase representa a los estudiantes que harán uso del laboratorio.

Laboratorio: El laboratorio es un aula acondicionada, dotada con los medios necesarios, para el desarrollo de clases prácticas y otros trabajos relacionados con la enseñanza. Este contiene de uno a muchos ejercicios prácticos.

Ejercicios Prácticos: Son parte fundamental del dominio del problema pues los ejercicios representan las diferentes prácticas que se pueden realizar en un laboratorio. Los ejercicios prácticos junto con los elementos teóricos son las bases para el proceso de enseñanza de un tópico en particular. Como ejemplos podemos mencionar, en un laboratorio de química, mezclar dos sustancias para observar el resultado o calentar una probeta para igualmente valorar el resultado.

Profesor: Es el encargado de dirigir el laboratorio además de impartir una evaluación a los ejercicios realizados por los estudiantes.

Objetivos: Los objetivos se deben enunciar en función del estudiante, de lo que éste debe ser capaz de lograr en términos de aprendizaje en el laboratorio en general o en los ejercicios prácticos en específico. En los objetivos deben evidenciarse las habilidades a lograr (acciones y operaciones) por parte de los estudiantes.

Sistema de evaluación: Representa la forma de calificar al estudiante por la realización del laboratorio. Un estudiante puede tener más de una evaluación, pero una evaluación pertenece a un único estudiante.

Evaluación: Representa la forma de calificar al estudiante por la realización del laboratorio. Un estudiante puede tener más de una evaluación y esta última pertenece a un único estudiante.

Herramientas: Las herramientas son todos aquellos recursos que ponen al alumno en contacto con la realidad.

2.2 Descripción general de la arquitectura

El proceso de desarrollo anterior permitió lograr una retroalimentación hacia esta AS y trató de convertir los procesos más comunes a todos los laboratorios virtuales en

componentes. Así, es posible ensamblar un cierto número de estos y lograr la funcionalidad básica de un laboratorio virtual a la vez que se reducen los costos de desarrollo.

A partir del estudio realizado sobre los elementos y necesidades ya mencionadas se propone utilizar como estilos arquitectónicos los basados en capas y en componentes. La combinación de estos estilos permitirá un amplio nivel de reutilización y facilidad de desarrollo, tal como se expone en las ventajas de estos estilos en el Capítulo 1.

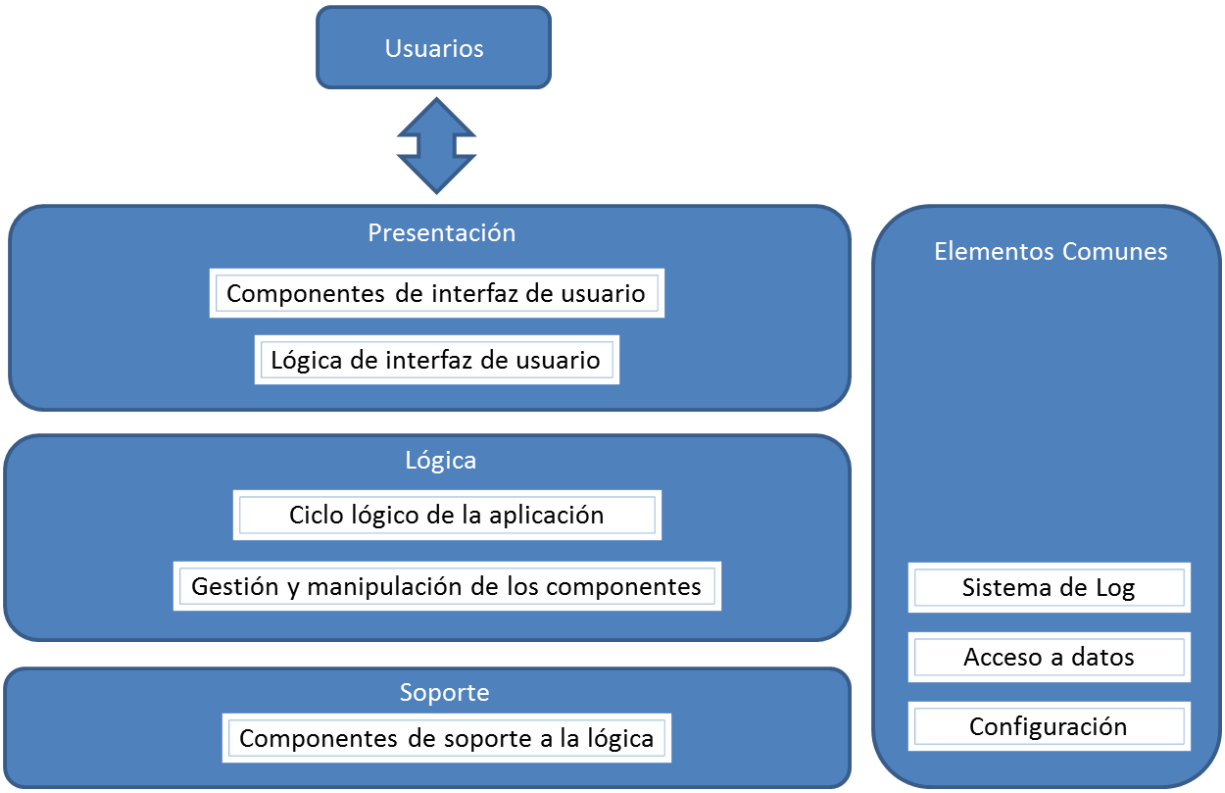

Figura 7. Organización de las capas.

En la figura 7, se muestran las diferentes capas de la arquitectura propuesta. La capa de presentación es la encargada de interactuar con el usuario y manejar los eventos que ocurran en esta. Igualmente, posibilita la incorporación de componentes de interfaz de usuario y permite reutilizar elementos comunes a otros laboratorios. La capa lógica es la encargada de ejecutar los requerimientos del negocio que son específicos a cada laboratorio virtual. Le sigue la capa de soporte que contendrá los componentes de soporte a la lógica de la aplicación que por sus características puedan ser utilizados por más de

un laboratorio virtual. En la figura 9 se aprecia también una capa transversal, Elementos Comunes. Esta tiene como objetivo proveer un conjunto de funciones que pueden ser accesibles desde cualquier capa. Esto simplifica la complejidad pues evita llamadas innecesarias a las capas adyacentes. Como parte también del estilo arquitectónico propuesto, cada capa mantiene comunicación solo con su capa adyacente lo que mejora el proceso de separación de las mismas en caso de ser necesario.

2.3 Herramientas de desarrollo

Las herramientas de desarrollo son las que de una forma u otra trazan guías y convenciones para desarrollar un *software*. La correcta selección, así como su dominio, representan importantes puntos a tener en cuenta para posibilitar un flujo continuo en el desarrollo del sistema.

2.3.1 Herramienta CASE[4] Visual Paradigm

Visual Paradigm es una de las herramientas CASE del mercado considerada como muy completa y fácil de usar, con soporte multiplataforma y que proporciona excelentes facilidades de interoperabilidad con otras aplicaciones. Puede ser utilizada durante todo el ciclo de desarrollo de software permitiendo la captura de requisitos, análisis, diseño e implementación.

Permite invertir código fuente de programas, archivos ejecutables y binarios en modelos UML[5] al instante. Está diseñada para usuarios interesados en sistemas de software de gran escala con el uso del acercamiento orientado a objeto. Además, apoya los estándares más recientes de las notaciones de Java y de UML.

2.3.2 Framework de desarrollo QT

Qt es un *framework* para el desarrollo de aplicaciones multiplataforma creado por la compañia Trolltech, actualmente propiedad de Nokia. Una de las funciones más conocidas de Qt es la de la creación de interfaces de usuario. Sin embargo no se limita a esto, ya que también provee varias clases para facilitar ciertas tareas de programación

[4] *Computer Aided Software Engineering*, en español Ingeniería de Software Asistida por Computadora
[5] *Unified Modeling Language*, en español Lenguaje de Modelado Unificado.

como el manejo de sockets, soporte para programación multi-hilo, comunicación con bases de datos, manejo de cadenas de caracteres, entre otras. Qt utiliza C++ de manera nativa, pero ofrece soporte para otros lenguajes como Python, Java, C#, Ruby entre otros.

Qt es un *framework* muy poderoso, comparable con Swing de Java o .NET de Microsoft, además ofrece una *suite* de aplicaciones para facilitar y agilizar las tareas de desarrollo, las aplicaciones que componen esta *suite* son:

Qt Assistant: Herramienta para visualizar 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++, pero especialmente diseñado para Qt, integra las primeras dos herramientas mencionadas.

Qt es utilizado por empresas como Intel, Google, Dreamworks y otras. Tiene una amplia documentación en línea además de una gran comunidad de desarrolladores. Cuenta actualmente con un sistema de dos licencias para su distribución y uso. La licencia libre GNU *Lesser General Public License* (LGPL) en su versión 2.1 y una versión comercial conocida como *Qt Commercial License.*

2.3.3 Motor gráfico OGRE3D[6]

OGRE3D es un motor gráfico 3D orientado a escena. Escrito en el lenguaje de programación C++ está diseñado de forma tal que posibilita desarrollar aplicaciones de manera sencilla e intuitiva usando las bondades del hardware gráfico del Pc. El sistema de clases de OGRE3D garantiza una abstracción a las API gráficas OpenGl y DirectX, además de proveer un conjunto de funcionalidades que agilizan el desarrollo. Con una activa comunidad de desarrollo. OGRE3D es una buena elección para proyectos multiplataforma y libre de uso.

[6] *Object-Oriented Graphics Rendering Engine,* en español motor gráfico orientado a objetos.

2.3.4 Control de versiones Subversion

Subversion es un software de sistema de control de versiones de código abierto y gratuito. Soporta el manejo de ficheros y directorios a través del tiempo utilizando un árbol de ficheros como repositorio central. Este repositorio funciona como un servidor de ficheros convencional, excepto que recuerda todos los cambios hechos a sus ficheros y directorios. Permite recuperar versiones antiguas de datos y examinar el historial de cambios de realizados en estos. Subversion puede acceder al repositorio a través de la red, lo que le permite ser usado por personas que se encuentran en distintos ordenadores. Existen varias interfaces de Subversion, tales como programas individuales o como interfaces que lo integran en entornos de desarrollo (35).

2.3.5 Lenguaje de Programación C++

C++ es un lenguaje de programación diseñado por Bjarne Stroustrup como extensión del lenguaje de programación C. Entre sus principales características se encuentra:

Programación orientada a objetos: La posibilidad de orientar la programación a objetos permite al programador diseñar aplicaciones desde un punto de vista más cercano a la vida real. Además permite la reutilización del código de una manera más lógica y productiva.

Portabilidad: Un código escrito en C++ puede ser compilado en diferentes ordenadores y sistemas operativos sin hacer apenas cambios.

Brevedad: El código escrito en C++ es muy corto en comparación con otros lenguajes, sobre todo porque en este lenguaje es preferible el uso de caracteres especiales que las "palabras clave".

Programación modular: Un cuerpo de aplicación en C++ puede estar hecho con varios ficheros de código fuente que son compilados por separado y después unidos. Esta característica permite unir código en C++ con código producido en otros lenguajes.

Velocidad: El código resultante de una compilación en C++ es muy eficiente, por su capacidad de actuar como lenguaje de alto y bajo nivel (36).

2.4 Requisitos Funcionales

Los requerimientos funcionales de un sistema describen lo que el sistema debe hacer. Estos requerimientos dependen del tipo de *software* que se desarrolle, de los posibles usuarios del *software* y del enfoque general tomado por la organización al redactar los requerimientos. En principio, la especificación de requisitos funcionales debe estar completa y consistente. La completitud significa que todos los servicios solicitados por los usuarios deben estar definidos. La consistencia significa que los requerimientos no deben tener definiciones contradictorias.

RF1 – Registrar usuario.

RF2 – Seleccionar Ejercicio.

RF3 – Cargar Ejercicio.

RF3.1 – Leer fichero .*scene*.

RF3.2 – Inicializar motor gráfico.

RF3.3 – Dibujar la información contenida en los ficheros .*scene*.

RF4 – Cambiar Ejercicio.

RF4.1 – Cerrar ejercicio en ejecución.

RF5 – Hacer reportes sobre el ejercicio realizado.

RF4 – Mostrar la ayuda.

2.5 Requisitos no funcionales

Los requisitos no funcionales son las cualidades o propiedades que el producto debe tener. En muchos casos los requisitos no funcionales son fundamentales en el éxito del producto. Debe pensarse en estas propiedades como las características que hacen al producto atractivo, usable, rápido o confiable. Están vinculados normalmente a requisitos funcionales, es decir, una vez que conozca lo que el sistema debe hacer es posible determinar cómo ha de comportarse, qué cualidades debe tener o cuán rápido o grande debe ser.

Como la AS propuesta va a ser usada para la creación de varios laboratorios virtuales los requisitos no funcionales se heredarán por los productos finales de forma obligatoria. De esta forma se impone cierta uniformidad en cada laboratorio virtual desarrollado con la AS propuesta y se aseguran algunos parámetros de calidad como la usabilidad y fiabilidad del software.

2.5.1 Usabilidad

El sistema deberá poseer una interfaz y navegación funcional, tanto para usuarios expertos, como para los que no tienen conocimientos profundos de Informática.

El sistema tendrá siempre visible la opción de ayuda posibilitando un mejor aprendizaje de las funcionalidades por parte de los usuarios.

El sistema deberá contar con combinaciones rápidas de teclas que faciliten el acceso a funcionalidades muy utilizadas del software para el uso de usuarios avanzados.

2.5.2 Hardware

El sistema necesitará como requisitos de *hardware* mínimos:

- Procesador Pentium 4 o equivalente
- 512 Mb de memoria RAM
- *Hardware* de video con 64 Mb de memoria.

Los requisitos de *hardware* recomendados son:

- Procesadores de doble núcleo o superior.
- 1 Gb de memoria RAM.
- *Hardware* de video con 256 Mb de memoria o superior y soporte para *Shaders* 3.0 o superior.

2.5.3 Soporte

El proceso de mantenimiento y soporte deberá ser automatizado, de forma tal que el cliente final solo tendrá que ejecutar las actualizaciones correspondientes y estas se encargarán de corregir, actualizar o incorporar elementos y funcionalidades a la aplicación final.

2.5.4 Software

Sistema operativo:

- Windows XP o superior.
- Linux en cualquiera de sus distribuciones basadas en Debian.

Últimos *drivers* compatibles con el *hardware* de la Pc, en especial con el de *hardware* de video.

2.6 Restricciones en el diseño e implementación

Como principal primicia del diseño y la implementación de la arquitectura deberá permitir la fácil integración o separación de componentes.

- Se utilizará el estándar de código “Estándares de codificación para el lenguaje C++ utilizado en el Centro de Diseño y Simulaciones de Estructuras Mecánicas versión 1.1”, para el desarrollo de aplicaciones sobre C++.

- El compilador de C++ que se utilizará para los *release* finales será MinGW para Windows y G++ para Linux.
- Como Entorno de Desarrollo Integrado (IDE) se podrá escoger opcionalmente Visual Studio 2008 ó QT Creator.
- Se utilizará la herramienta Case Visual Paradigm for UML 2.0 para el diseño y modelado.
- Se utilizará las siguientes versiones para desarrollar los componentes y la AS:
  OGRE3D versión 1.7.3
  QT *framework* versión de escritorio 4.7.3
- Todas las Clases pertenecientes al *namespace* "coreVLab" deberán estar marcadas con la directiva MyExport[7].

2.7 Estrategia de implementación

El propio desarrollo basado en componentes propone una estrategia de implementación bien definida. Esta estrategia sin dudas representa una gran ayuda a la hora de construir aplicaciones muy parecidas con elementos claramente reutilizables como es el caso de los laboratorios virtuales a desarrollar.

Como parte de la estrategia de implementación se proponen los siguientes pasos para el desarrollo de cada componente:

- Se selecciona la funcionalidad que por sus características puede ser reutilizable a otros laboratorios virtuales.
- Se determina la capa de la aplicación en la cual está presente la funcionalidad.
- Se implementa el componente teniendo en cuenta las restricciones de la capa.
- Se prueba el componente y sus funcionalidades básicas como unidad.
- Se prueba el componente y sus funcionalidades acopladas a la AS.

[7] Esta directiva indica a los compiladores que el enlace a la hora de compilación se efectuará con una biblioteca externa.

- Se documenta y se ubica el componente en el repositorio del proyecto, para su utilización por los desarrolladores.

El componente como unidad básica deberá cumplir los siguientes términos:

- Ser usado por los desarrolladores de los laboratorios virtuales sin la intervención de su creador.
- Incluye una especificación de sus dependencias.
- Incluye una especificación de la funcionalidad que ofrece.
- Se acopla a la AS de manera rápida y sin complicaciones.

Se puede apreciar que utilizando esta estrategia de desarrollo se pueden identificar dos procesos fundamentales, la creación de componentes y la integración de los mismos en la AS. El aseguramiento de un buen repositorio de componentes, agiliza el desarrollo de varios laboratorios virtuales de forma paralela. Igualmente, da cumpliendo a las necesidades de uniformidad. Los mayores esfuerzos de implementación están centrados en la lógica de la aplicación reduciendo los tiempos totales de desarrollo.

2.8 Vistas arquitectónicas

Siguiendo los pasos que propone la metodología RUP para describir la AS, utilizando el modelo "4+1" de Kruchten, mencionado en el Capítulo 1. Proponemos las siguientes vistas:

La vista de casos de uso o escenarios: es una representación de los casos de uso que tienen mayor importancia para la arquitectura.

La vista lógica: que comprende las abstracciones fundamentales del sistema a partir del dominio del problema.

La vista de proceso: describe los aspectos de concurrencia y sincronización del diseño.

La vista de despliegue: un mapeado del *software* sobre el *hardware*.

La vista de implementación: la organización estática de módulos en el entorno de desarrollo.

El objetivo de las vistas de la arquitectura es la simplificación o abstracción de los modelos, de los cuales se destacan los detalles más significativos y se obvian los que no representan un aporte significativo a la visión de la solución del problema.

2.8.1 Vista de casos de uso

Cada caso de uso describe una o varias funcionalidades que el sistema debe cumplir. Existe una interacción entre los casos de uso y la arquitectura, los casos de uso deben encajar en la arquitectura cuando se llevan a cabo y la arquitectura debe permitir el desarrollo de todos los casos de usos requeridos, actuales y del futuro. Esto provoca que ambos deban evolucionar en paralelo durante todo el proceso de desarrollo de software.

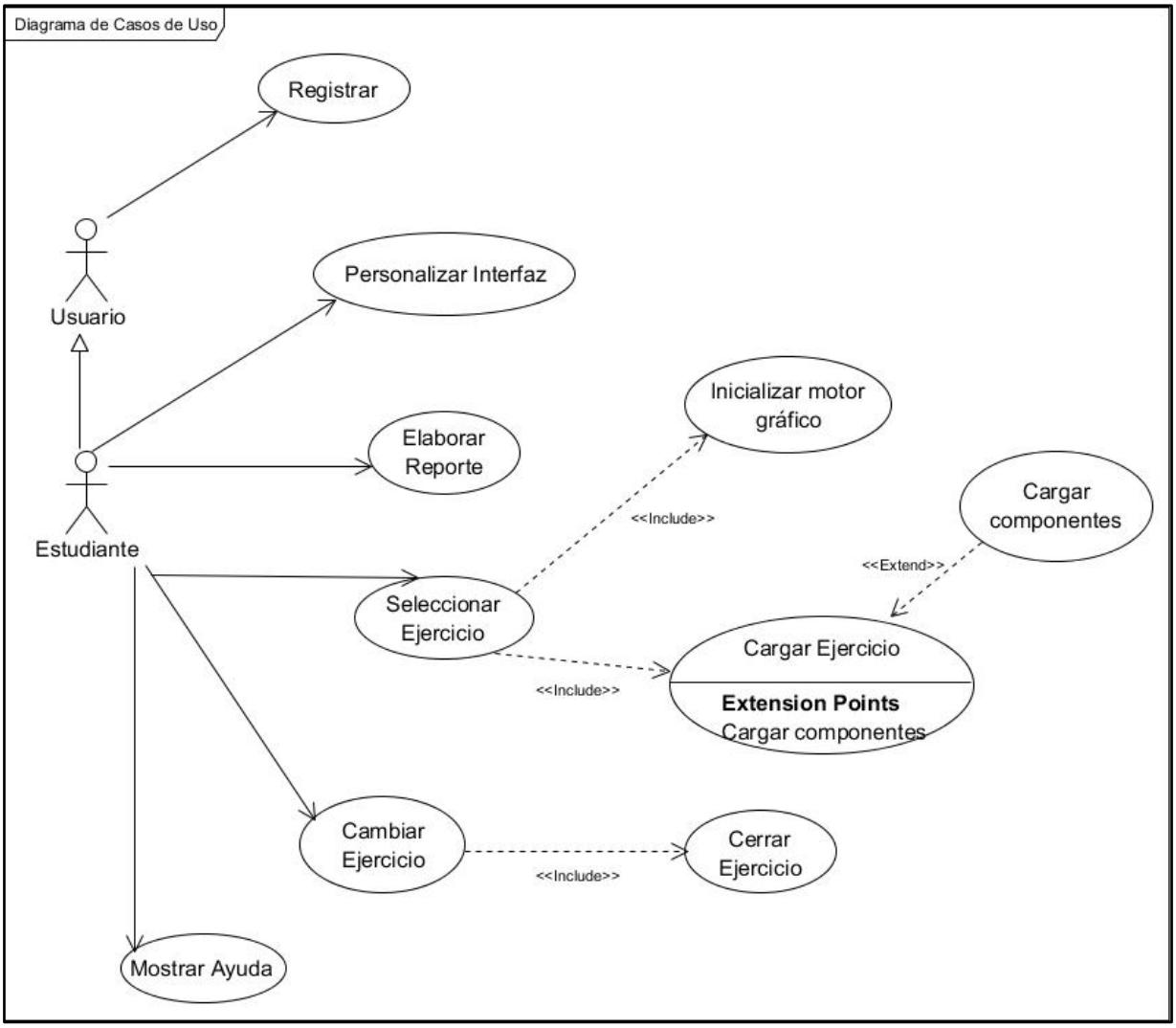

Figura 8. Diagrama de Casos de Uso del sistema.

A continuación se describen brevemente los casos de uso:

Caso de uso Registrarse: Solicita al usuario sus datos para guardar registro de uso.

Caso de uso Elaborar Reporte: Permite al usuario crear un reporte del trabajo realizado en el laboratorio virtual para su posterior evaluación por parte de los profesores.

Caso de uso Mostrar Ayuda: Facilita al usuario los manuales de uso de la aplicación además de otros temas que puedan resultar de interés.

Caso de uso Personalizar interfaz: Permite al usuario personalizar la disposición de los elementos en la aplicación.

Caso de uso Seleccionar Ejercicio: Permite al usuario seleccionar y ejecutar el ejercicio de su preferencia, este caso de uso para su correcta realización se apoya en los casos de uso Iniciar motor gráfico y Cargar ejercicio.

Caso de uso Inicializar motor gráfico: Este caso de uso se encarga de todas las tareas de inicialización y configuración del motor gráfico OGRE3D así como su puesta en marcha en conjunto con el *framework* QT.

Caso de uso Cargar ejercicio: Carga la biblioteca asociada al ejercicio y gestiona los eventos asociados al mismo.

Caso de uso Cargar componentes: Gestiona la carga de los componentes que sirven de apoyo para la realización del ejercicio.

Caso de uso Cambiar de ejercicio: Permite al usuario que está realizando un ejercicio cambiar para otro ejercicio. La realización de este caso de uso se apoya en el caso de uso Cerrar ejercicio.

Caso de uso Cerrar ejercicio: Cierra el ejercicio actual, limpiando la memoria asociada al uso de este.

2.8.2 Vista Lógica

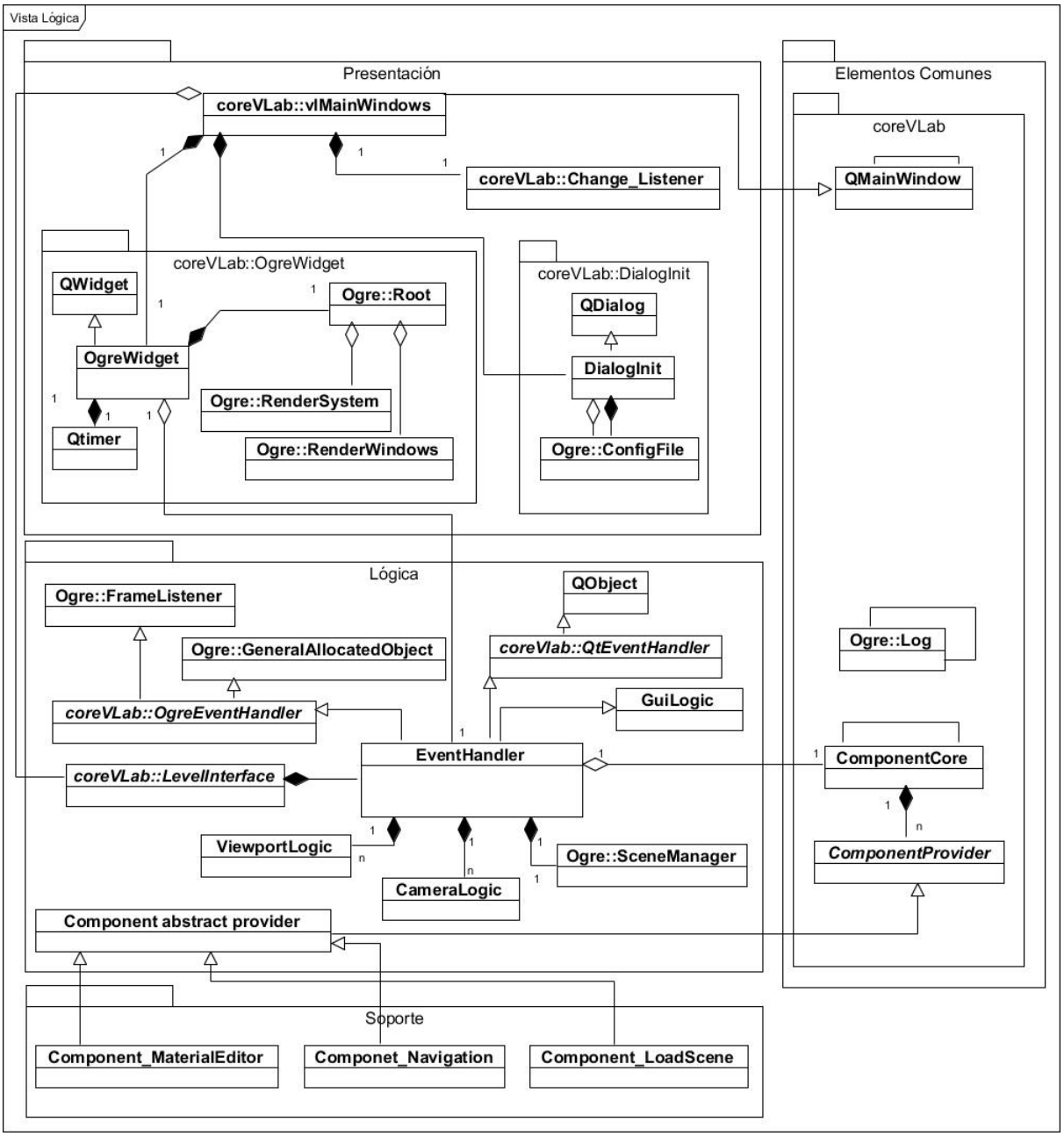

Figura 9. Vista Lógica.

En la figura 9 se pueden ver las diferentes capas, así como las clases que contiene cada una de ellas. La capa de presentación contiene los componentes de la presentación. Todos estos componentes tienen en común que heredan de clases del *framework* QT, que son parte de la interfaz gráfica. Ejemplo: *QWidget* y *QDialog*. Igualmente, en esta

capa, ocurre el proceso de carga del componente dinámico que contiene el ejercicio a través de la clase interfaz *coreVLab::LevelInterface*. La clase *coreVLab::vlMainWindows* es la encargada de contener todos los compontes de interfaz gráfica así como controlar la distribución física de estos en la ventana de la aplicación.

La capa lógica contiene toda la realización de un ejercicio. Toda esta capa está encapsulada en un solo componente dinámico. Lo que permite que se pueda cambiar de lógica en caso de que la práctica de laboratorio contemple más de un ejercicio.

La capa de soporte contiene los componentes dinámicos que sirven de apoyo a la realización de un ejercicio determinado. El control de estos componentes radica en la capa lógica.

La capa elementos comunes contiene las clases que pueden ser accesibles desde cualquier punto de la aplicación. La clase *ComponentCore* es la encarda de la administración de los componentes dinámicos que se encuentran en la capa de soporte. Contiene también la clase *ComponentProvider*. Esta es la interfaz abstracta a todos los componentes dinámicos de la capa de soporte. Igualmente, cada componente de este tipo tiene otra interfaz abstracta que expone sus especificaciones. Necesario para que en la capa lógica se conozca las particularidades de un componente en particular.

2.8.3 Vista de procesos

La vista de procesos provee una idea general de cómo ocurren las tareas en los diferentes hilos de la aplicación así como su sincronización. En el caso de la AS propuesta para laboratorios virtuales podemos representar dos procesos fundamentales que ocurren en la aplicación, ambos hilos son mutuamente excluyentes por lo que no necesitan ser sincronizados. Vale mencionar que tanto el *Framework* QT como el motor gráfico Ogre3D utilizan la implementación de sus funciones, pero su tratamiento es totalmente trasparente al desarrollador aunque esto último no significa que no se pueda cambiar.

Eventos de usuario: Este hilo es totalmente controlado por el *Framework* QT, procesa todas las entradas de usuario y las re-direcciona como eventos a los componentes.

Bucle de Pintado: Este hilo representa el continuo proceso de pintado del motor gráfico Ogre3D controlado por el desarrollador utilizando el componente *QTimer* del *Framework* QT.

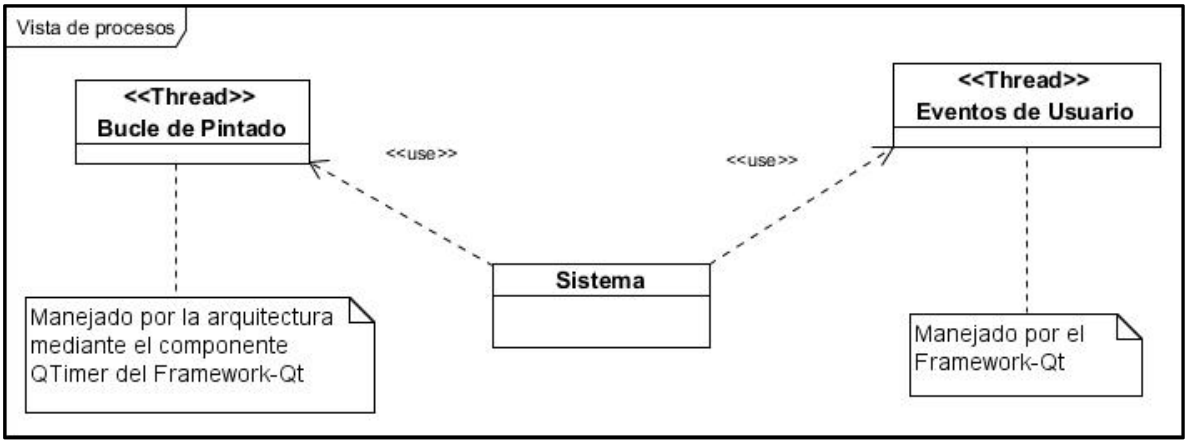

Figura 10. Vista de Procesos.

2.8.4 Vista de despliegue

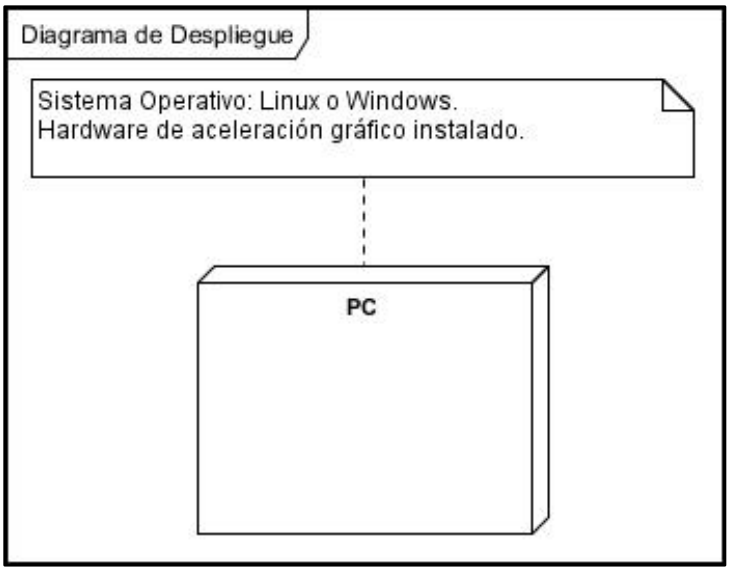

Figura 11. Vista de Despliegue

2.8.5 Vista de implementación

La vista de implementación nos proporciona una perspectiva general de los componentes más importantes para la arquitectura así como sus interfaces.

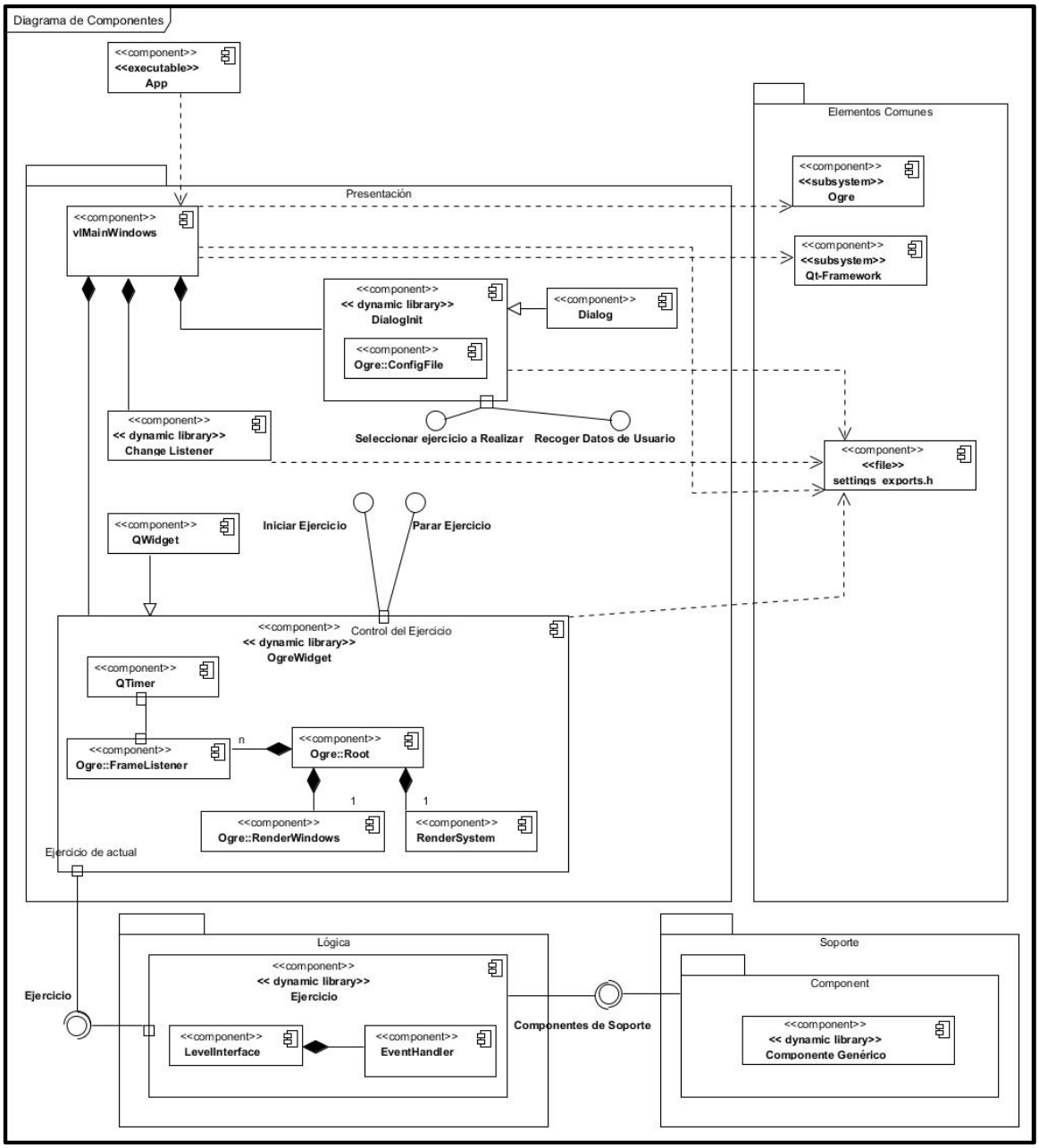

Figura 12. Vista de Implementación

2.9 Descripción de algunos componentes asociados a la arquitectura

Como parte fundamental de la arquitectura se describirán los principales componentes asociados a esta. Para cada componente se incluirá una ficha técnica con la información más relevante asociada a este así como su función en la arquitectura. A continuación se detallan los principales conceptos a tratar en la ficha técnica:

Capa en que se encuentra (CP): referido a la capa en que se encuentra el componente.

Dependencias (DP): referido a si el componente tiene dependencias directas con otros componentes.

Función principal dentro de la arquitectura (FPA): referido a las funciones que realiza dentro de la arquitectura así como a los requisitos funcionales a los que responde el componente:

Tipo de componte (TC): referido a la forma en que el componente se encuentra dentro de la arquitectura y como ocurre su enlazado. Por ejemplo: componente estático o componente dinámico.

2.9.1 Componente OgreWidget

Ficha técnica componente *OgreWidget*	
CP	Capa de Presentación
DP	Ogre3d, Framework-Qt
FPA	El componente es el encargado de controlar el estado de ejecución del motor gráfico OGRE3D. También se encarga de la manipulación del tiempo para la visualización del *render*. Este componente tiene un peso fundamental en la arquitectura pues es el encargado de visualizar los contenidos a mostrar por los diferentes ejercicios. Forma parte del *namespace* “coreVLab” que contiene todos los componentes indispensables para iniciar un laboratorio virtual.

TC	Componente estático, distribuido en forma de biblioteca dinámica.

Tabla 1. Ficha Técnica Componente OgreWidget.

2.9.2 Componente DialogInit

Ficha técnica componente *DialogInit*	
CP	Capa de Presentación
DP	Ogre3d, Framework-Qt
FPA	El componente es el encargado de registrar al usuario al iniciar el Laboratorio Virtual además de permitir la selección de los posibles ejercicios a realizar en la aplicación. Forma parte del *namespace* “coreVLab” que contiene todos los componentes indispensables para iniciar un laboratorio virtual.
TC	Componente estático, distribuido en forma de biblioteca dinámica.

Tabla 2. Ficha técnica componente DialogInit.

2.9.3 Componente EventHandler

Ficha técnica componente *EventHandler*	
CP	Capa Lógica
DP	Ogre3d, Framework-Qt, LevelListener
FPA	EventHandler representa un ejercicio para el laboratorio virtual. Este maneja todo los eventos relacionados al ejercicio, y mantiene estrecha comunicación con los componentes de soporte a la lógica. Es manipulado por la arquitectura como componente dinámico de forma tal que se puedan incorporar o remplazar de una manera rápida y sencilla. Forma parte del *namespace* “coreVLab” que contiene

	todos los componentes indispensables para iniciar un laboratorio virtual.
TC	Componente dinámico, distribuido en forma de biblioteca dinámica.

Tabla 3. Ficha técnica componente EventHandler.

2.9.4 Componente LoadScene

Ficha técnica componente *LoadScene*	
CP	Capa Soporte
DP	Ogre3d, Framework-Qt, ComponentProvider, ComponentCore
FPA	El componente LoadScene se encarga de cargar los ficheros ".scene" que contienen la información de configuración de los objetos que conforman un laboratorio virtual. También el componente informa sobre el estado de carga de cada elemento.
TC	Componente dinámico, distribuido en forma de biblioteca dinámica.

Tabla 4. Ficha técnica componente LoadScene.

2.10 Patrones de diseño utilizados

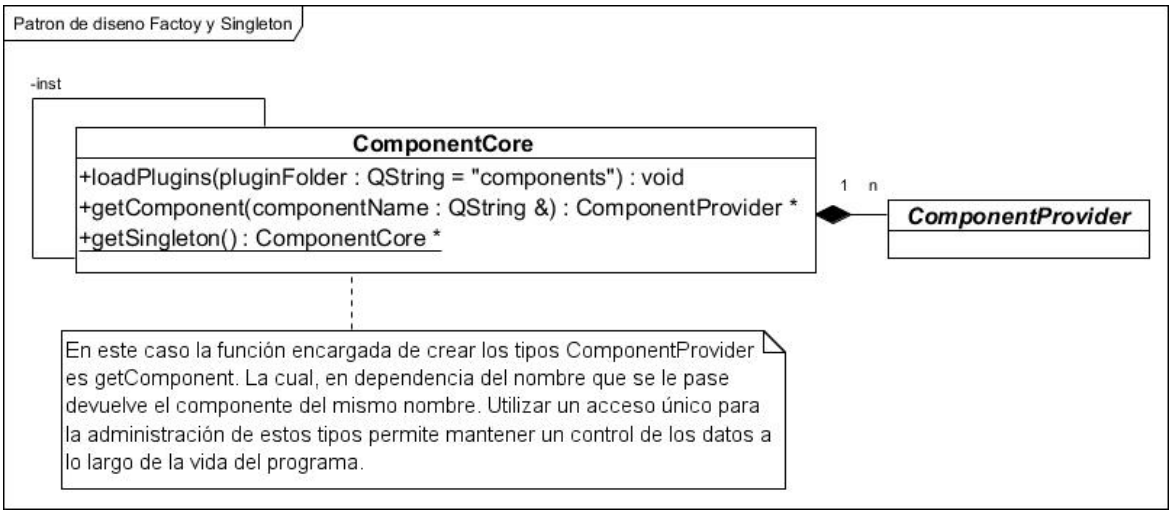

Figura 13. Aplicación del patrón de diseño *Singleton* en conjunto con el *Factory Method*

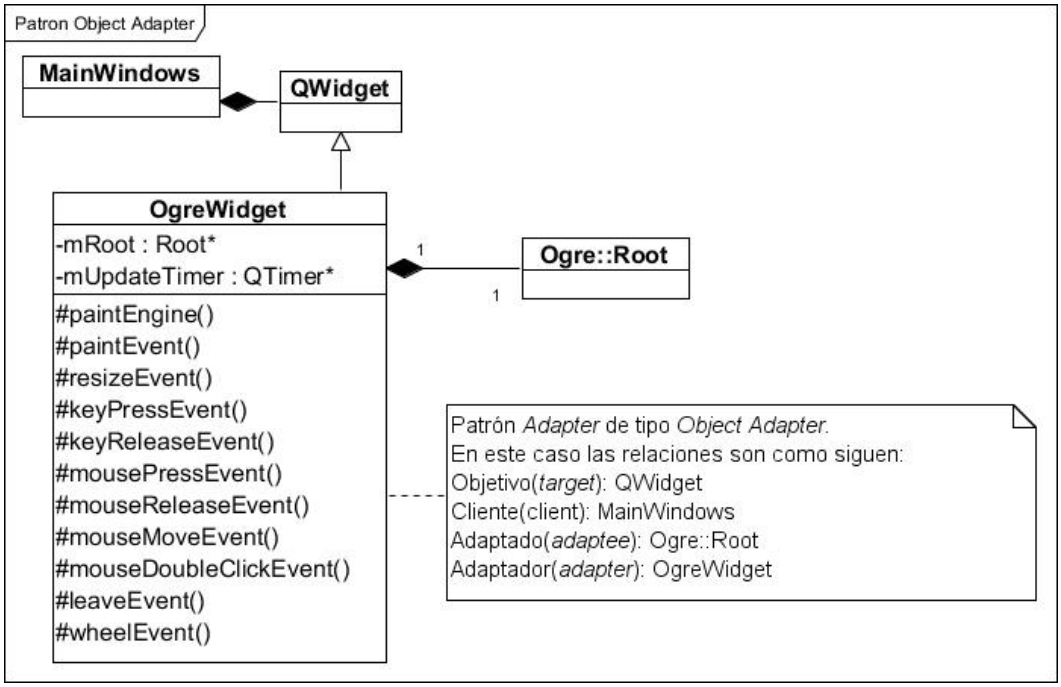

Figura 14. Aplicación del patrón de diseño *Adapter*

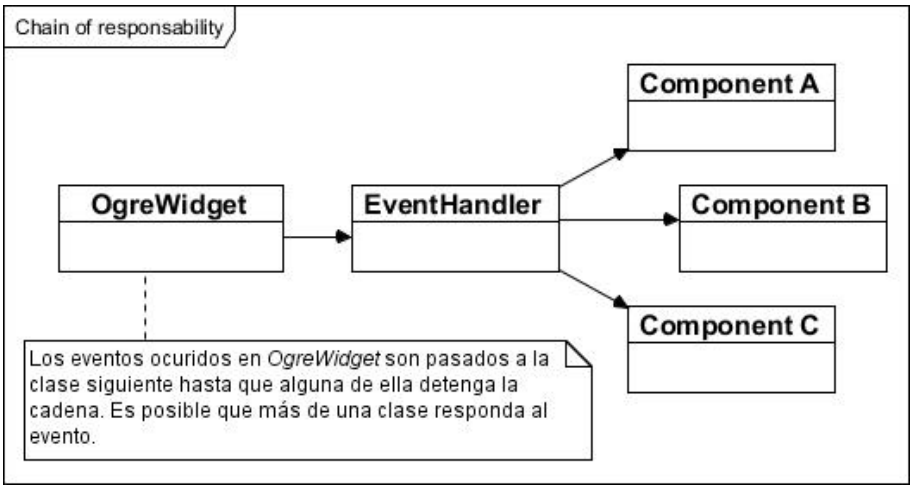

Figura 15. Aplicación del patrón de diseño *Chain of Responsibility*

2.11 Discusión general de la arquitectura

En el Capítulo 1, sección 1.9, se describen los problemas que presentaba la arquitectura anterior. A continuación se detalla la forma en que la arquitectura propuesta soluciona algunos de estos problemas.

- Portabilidad: La ventana utilizada en la arquitectura anterior para representar contenidos heredaba su configuración a partir de la clase *QGLWidget* perteneciente al *framework* QT. Esto traía como consecuencia que el motor gráfico Ogre3D no podía escribir la configuración adecuada para Opengl en Windows, y la aplicación terminaba inesperadamente. Este problema no ocurría en Linux, por lo que se crearon diferentes implementaciones para cada sistema operativo. En la arquitectura propuesta no se utiliza como base la clase *QGLWidget*. En cambio se hereda de la clase *QWidget* y se configura de forma manual todos los parámetros necesarios para la inicialización de Ogre3D. Como resultado la implementación solo posee cambios menores dependiendo del sistema operativo a utilizar.

- Configurabilidad: La incorporación a la arquitectura de un componente de configuración permitirá aprovechar al máximo las posibilidades del *hardware* existente.
- Modificabilidad: Los componentes al ser tratados como elementos independientes se pueden actualizar sin influir en otros elementos de la arquitectura.
- Mantenibilidad: La arquitectura propuesta tiene bien definida cada una de sus capas y la lógica a seguir para dar solución a un problema determinado.

Existen temas importantes que se deben tener en cuenta respecto al diseño propuesto. Estos son:

- Alto acoplamiento al *framework* Qt. Resulta muy difícil pensar en la arquitectura propuesta sin el uso del *framework* Qt. La mayoría de los componentes están realizados sobre esta base. Además que la garantía de ser un sistema portable pertenece completamente al *framework* Qt.
- No existe un mecanismo para adicionar funcionalidades que no esté controlado por los desarrolladores. Por ejemplo: no se puede añadir una nueva funcionalidad

a un ejercicio determinado, a no ser que el ejercicio sea actualizado por el desarrollador y luego se remplace por la versión anterior a este.

- Gran parte de los componentes en la arquitectura son componentes dinámicos. Esto provoca la existencia de un tiempo asociado a la carga y el chequeo de estos. Esto ocurre cuando la aplicación ya esta iniciada y aunque no afecta el rendimiento de forma drástica, debe ser un elemento a analizar para posteriores iteraciones.

CAPÍTULO 3: EVALUACIÓN DE LA ARQUITECTURA

La evaluación de la arquitectura es un paso fundamental para asegurar la calidad del *software* desarrollado, y es que atributos como disponibilidad, integrabilidad, modificabilidad y portabilidad están estrechamente vinculados con la AS.

La AS es de los primeros artefactos obtenidos en la fase de diseño por lo que su evaluación provee indicadores que permiten la oportunidad de resolver problemas que pueden presentarse a nivel arquitectónico.

Independientemente de la metodología utilizada, la intención de evaluar la AS es, asegurar que el sistema cumpla con los servicios y la funcionalidad que espera el usuario. De igual forma, debe asegurar los atributos de calidad que deben cumplirse, y que dirigen las decisiones en el momento de la construcción de la arquitectura del sistema.

3.1 Cuando evaluar la arquitectura de software

Si bien es posible evaluar la arquitectura en cualquier fase del proceso de desarrollo de *software* existen dos variantes convenientemente definidas de cuándo hacer la evaluación: evaluación temprana y evaluación tardía.

Evaluación temprana: es aquella que no tiene que esperar a que la arquitectura esté totalmente especificada. Esta puede ser realizada desde fases tempranas y a lo largo del proceso de desarrollo, para examinar las decisiones arquitectónicas ya tomadas y decidir entre las opciones que están pendientes (37).

Evaluación tardía: consiste en realizar la evaluación de la arquitectura cuando se encuentra establecida y la implementación se ha completado. Este es el caso general que se presenta en el momento de la adquisición de un sistema ya desarrollado. Se considera muy útil la evaluación del sistema en este punto, puesto que puede observarse el cumplimiento de los atributos de calidad asociados al sistema, y cómo será su comportamiento general (37).

La evaluación de una arquitectura de software es una tarea no trivial, puesto que se pretende medir propiedades del sistema en base a especificaciones abstractas, como por ejemplo los diseños arquitectónicos. Por ello, la intención es la evaluación del potencial de la arquitectura diseñada para alcanzar los atributos de calidad requeridos. Las mediciones que se realizan sobre una arquitectura de software pueden tener distintos objetivos, dependiendo de la situación en la que se encuentra el arquitecto y la aplicabilidad de las técnicas que emplea (38).

Es importante destacar que la evaluación no define si una arquitectura determinada es buena o no, simplemente expresa donde se encuentran los riesgos y fortalezas de la misma.

3.2 Atributos de calidad y su relación con la arquitectura

Los atributos de calidad no son más que los requerimientos adicionales del sistema, que hacen referencias a características que este debe satisfacer. En la metodología RUP estos atributos están relacionados con los requisitos no funcionales de un sistema.

Estos se clasifican en dos tipos: observables en vías de ejecución y no observables en vías de ejecución.

Observables mediante la ejecución: aquellos atributos que se determinan del comportamiento del sistema en tiempo de ejecución. Entre ellos se encuentran:

- Disponibilidad: Es la medida de disponibilidad del sistema para el uso.
- Confidencialidad: Es la ausencia de acceso no autorizado a la información.
- Funcionalidad: Habilidad del sistema para realizar el trabajo para el cual fue concebido.
- Desempeño: Es el grado en el cual un sistema o componente cumple con sus funciones designadas, dentro de ciertas restricciones dadas, como velocidad, exactitud o uso de memoria.

- Confiabilidad: Es la medida de la habilidad de un sistema a mantenerse operativo a lo largo del tiempo.

- No observables en la ejecución: aquellos atributos que se establecen durante el desarrollo del sistema. Entre los que podemos mencionar:

- Configurabilidad: Posibilidad que se otorga a un usuario experto a realizar ciertos cambios al sistema.

- Integrabilidad: Es la medida en que trabajan correctamente componentes del sistema que fueron desarrollados separadamente al ser integrados.

- Integridad: Es la ausencia de alteraciones inapropiadas de la información.

- Interoperabilidad: Es la medida de la habilidad de que un grupo de partes del sistema trabajen con otro sistema.

- Modificabilidad: Es la habilidad de realizar cambios futuros al sistema.

- Mantenibilidad: Capacidad de modificar el sistema de manera rápida y a bajo costo.

- Portabilidad: Es la habilidad del sistema para ser ejecutado en diferentes ambientes de computación. Estos ambientes pueden ser *hardware*, *software* o una combinación de ambos.

- Reusabilidad: Es la capacidad de diseñar un sistema de forma tal que su estructura o parte de sus componentes puedan ser reutilizados en futuras aplicaciones.

Los requerimientos de calidad se ven altamente influenciados por la arquitectura del sistema, su calidad debe ser considerada en todas las fases del diseño, pero los atributos de calidad se manifiestan de maneras distintas a lo largo de estas fases. De esta forma, establecen que la arquitectura determina ciertos atributos de calidad del sistema, pero existen otros atributos que no dependen directamente de la misma (5).

Relacionar los atributos de calidad de un sistema a su arquitectura provee una base para la toma de decisiones objetivas sobre acuerdos de diseño y permite a los ingenieros realizar predicciones razonablemente exactas sobre los atributos del sistema. El objetivo de fondo es lograr la habilidad de evaluar cuantitativamente y llegar a acuerdos entre múltiples atributos de calidad para alcanzar un mejor sistema de forma global.

3.3 Modelos de Calidad

Las diferentes formas de organizar y descomponer los atributos de calidad son conocidas como modelos de calidad, resultan de utilidad para la predicción de confiabilidad y en la gerencia de calidad durante el proceso de desarrollo.

Variadas han sido las propuestas desde los años 70 hasta la actualidad, pudiéndose mencionar los siguientes modelos:

- Modelo de McCall.
- Modelo FURPS.
- Modelo de Dromey.
- Modelo de calidad ISO/IEC 9126.
- ISO/IEC 9126 adaptado para arquitecturas de software.

Este último es una adaptación del Modelo de calidad ISO/IEC 9126 para efectos de la evaluación de AS. El modelo se basa en los atributos de calidad que se relacionan directamente con la arquitectura: funcionalidad, confiabilidad, eficiencia, mantenibilidad y portabilidad eliminando el atributo usabilidad propuesto por el modelo original pues este está relacionado con los componentes de interfaz de usuario que generalmente no son elementos significativos para una arquitectura.

Característica	Sub-característica	Elementos de tipo arquitectónico

Funcionalidad	Adecuación	Refinamiento de los diagramas de secuencia.
	Exactitud	Identificación de los componentes con las funciones responsables de los cálculos.
	Interoperabilidad	Identificación de conectores de comunicación con sistemas externos.
	Seguridad	Mecanismos o dispositivos que realizan explícitamente la tarea.
Confiabilidad	Tolerancia a fallas	Existencia de mecanismos o dispositivos de software para manejar excepciones
	Recuperabilidad	Existencia de mecanismos o dispositivos de software para restablecer el nivel de desempeño y recuperar datos.
Eficiencia	Desempeño	Componentes involucrados en un flujo de ejecución para una funcionalidad.
	Utilización de recursos	Relación de los componentes en términos de espacio y tiempo.
	Modularidad	Número de componentes que dependen de un componente.
Portabilidad	Adaptabilidad	Presencia de mecanismos

		de adaptación.
	Instalabilidad	Presencia de mecanismos de instalación.
	Coexistencia	Presencia de mecanismos que faciliten la coexistencia.
	Reemplazabilidad	Lista de componentes reemplazables para cada componente.

Tabla 5. Atributos de calidad planteados por Losavio. (39)

3.4 Técnicas de evaluación

Las técnicas de evaluación de arquitecturas de software pueden clasificarse en dos vertientes fundamentales las cuantitativas y las cualitativas (Ver Fig. 16). Generalmente las técnicas de evaluación cualitativas son utilizadas cuando la arquitectura está en construcción, mientras que las técnicas de evaluación cuantitativas, se usan cuando la arquitectura ya ha sido implantada. Estos tipos de técnicas posibilitan evaluar y establecer comparaciones entre arquitecturas candidatas para determinar cuál satisface más un atributo de calidad específico.

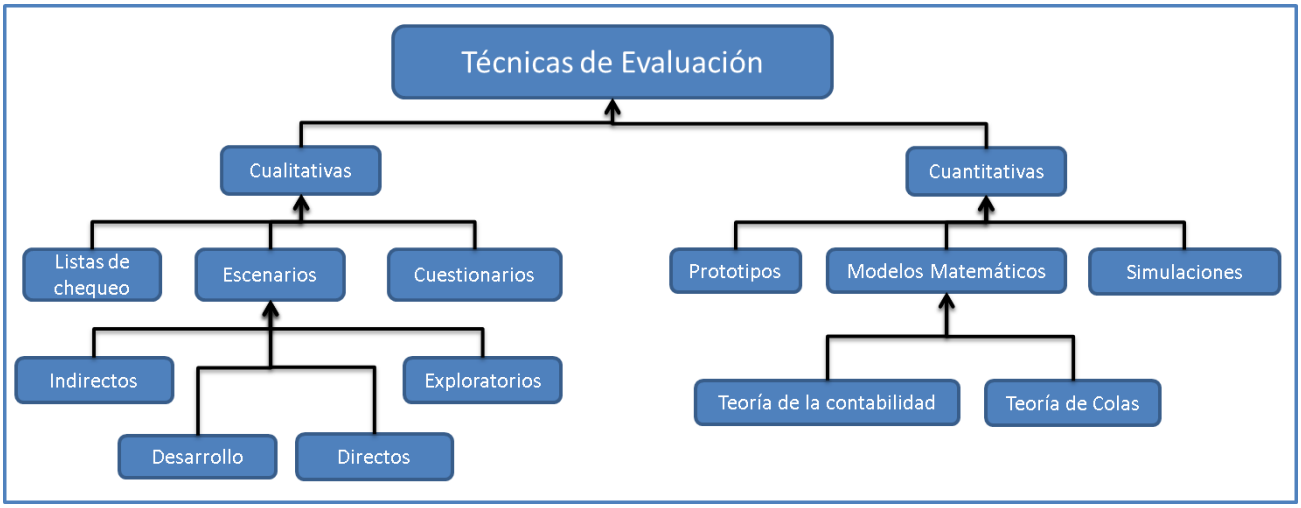

Figura 16. Técnicas de Evaluación.

Para la evaluación de la arquitectura propuesta se selecciona dentro de las técnicas cualitativas, la basada en escenarios teniendo en cuenta que tiene la ventaja de ser simple de crear y entender; no requiere mucho entrenamiento y son efectivas. A continuación se explica esta técnica.

3.4.1 Evaluación basada en escenarios:

Un escenario es una breve descripción de la interacción de alguno de los involucrados en el desarrollo del sistema con este. 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 la ejecución de tareas, cambios en el sistema, ejecución de pruebas, 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 es el que permite establecer cuál es el atributo de calidad asociado (37).

Dentro de las herramientas utilizadas por este método se encuentra el árbol de utilidad, que es un esquema en forma de árbol que presenta los atributos de calidad de un sistema de *software*.

3.5 Métodos de evaluación de arquitecturas de software

No fue hasta hace unos años, con el aumento del estudio de las AS, que aparecieron los primeros métodos para la evaluación de arquitecturas, pues los existentes tenían enfoques incompletos y no repetibles para cualquier AS. De forma general, los métodos permiten la búsqueda de conflictos en la arquitectura evaluando los atributos de calidad y utilizando las técnicas de evaluación. A continuación se explican varios de ellos.

3.5.1 SAAM

El método SAAM (*Software Architecture Analysis Method*) fue primero en ser ampliamente promulgado y documentado. El método fue originalmente creado para el análisis de la modificabilidad de una arquitectura, pero en la práctica ha demostrado ser

muy útil para evaluar de forma rápida distintos atributos de calidad, tales como modificabilidad, portabilidad, escalabilidad e integrabilidad.

El método de evaluación SAAM se enfoca en la enumeración de un conjunto de escenarios que representan los cambios probables a los que estará sometido el sistema en el futuro. Con la aplicación de este método se obtienen los lugares en que la AS puede fallar, en términos de los requerimientos de modificabilidad.

3.5.2 ATAM

El método ATAM (*Architecture Trade off Analysis Method*), está inspirado en tres áreas distintas: los estilos arquitectónicos, el análisis de atributos de calidad y el método de evaluación SAAM. Es por ello que este se concentra en la identificación de los estilos arquitectónicos presente en la AS a evaluar. Además, parte de la idea que los estilos arquitectónicos representan los medios empleados por la arquitectura para alcanzar los atributos de calidad, así como que permiten describir la forma en la que el sistema puede crecer, responder a cambios, integrarse con otros sistemas entre otros (37).

El método de evaluación ATAM comprende nueve pasos que se enumeran a continuación:

- Presentación de ATAM.
- Presentación de las metas del negocio.
- Presentación de la arquitectura.
- Identificación de los estilos arquitectónicos.
- Generación del árbol de utilidad.
- Análisis de los estilos arquitectónicos.
- Establecimiento de la prioridad a los escenarios.

- Análisis de los estilos arquitectónicos usando como base los escenarios.
- Presentación de resultados.

3.5.3 ARID

El método 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, completitud de la documentación y la conveniencia de los servicios que provee el diseño propuesto.

3.6 Evaluación de la arquitectura propuesta

Para la evaluación de la arquitectura propuesta se utilizará el modelo de calidad ISO/IEC 9126 adaptado para AS en conjunto con el método ATAM y la técnica de evaluación basada en escenarios.

ATAM es un método para identificar riesgos. Este debe ser ejecutado tempranamente en el ciclo de desarrollo del *software*, con bajos costes y de manera rápida, dependiendo en gran medida del nivel de especificación de la arquitectura desde su inicio. Debido a esto no es posible predecir exactamente cómo será el comportamiento de un atributo de calidad, pero si se podrá identificar los posibles riesgos asociados a beneficiar o descartar un atributo de calidad u otro.

Como se describe en la presentación del método ATAM cuenta con 9 pasos, los cuales se adaptaran de acuerdo a los requerimientos del proyecto.

3.6.1 Creación del árbol de utilidad

La creación de los escenarios se debe priorizar, ya que así los arquitectos podrán contar con más orientación para tomar futuras decisiones en base a la prioridad de estos y el resultado de su evaluación. Para el proceso de asignación de prioridades a los escenarios se tuvo en cuenta los siguientes puntos:

¿Qué pasaría si este escenario no se cumple?

¿Es posible compensar el no responder a este escenario?

¿Cuánto esfuerzo es necesario para lograr cumplir con el escenario?

Atributo	Sub-característica	Escenario	Prioridad
Funcionalidad	Seguridad	La integridad de los ficheros *media* está asegurada.	Baja
Confiabilidad	Tolerancia a fallas	Al ocurrir un error en la aplicación, es reportado por el sistema de log.	Alta
	Recuperabilidad	El sistema es capaz de volver al último estado estable de la aplicación después de ocurrir un error.	Media
Mantenibilidad	Facilidad de adaptación al cambio	El sistema debe permitir, después de desplegado, adicionar nuevos ejercicios sin muchas complicaciones.	Media
	Facilidad de adaptación al cambio	Mantenimiento de algún componente funcional.	Alta
Portabilidad	Adaptabilidad	El sistema puede ser portado en varios sistemas operativos.	Alta

Tabla 6. Árbol de Utilidad.

3.6.2 Análisis de los estilos arquitectónicos

Como se expone en el Capítulo 2, los estilos arquitectónicos presentes son:

- Estilo basado en componentes.
- Estilo basado en capas.

Estos estilos promueven atributos como la reutilización, la separación de responsabilidades, la facilidad de instalación y la mantenibilidad que en definitiva son los que contribuyen a dar cumplimiento al objetivo principal del trabajo.

3.6.3 Especificación de los escenarios

Las tablas desde la 6 a la 11 muestran el tratamiento de los escenarios a través del método ATAM como parte del proceso de evaluación de la arquitectura.

Escenario #1	**La integridad de los ficheros media está asegurada**
Atributo(s)	Funcionalidad – Seguridad.
Estímulo	Se modifica una imagen contenida en la carpeta media de la aplicación.
Respuesta	El sistema debería mostrar la imagen con la modificación realizada.
Explicación	La arquitectura no tiene un mecanismo para asegurar la integridad de los ficheros media usada por la aplicación. Un usuario con los conocimientos adecuados puede cambiar estos ficheros y como consecuencia afectar la aplicación.

Tabla 7. Escenario 1.

Escenario #2	**Al ocurrir un error en la aplicación es reportado por el sistema de log.**
Atributo(s)	Confiabilidad – Tolerancia a fallas.

Estímulo	Se elimina una biblioteca de carga necesaria para la aplicación.
Respuesta	En el archivo Ogre.log se deben mostrar las trazas de la aplicación, entre ellos la traza asociada a la falta de elementos necesarios para ejecutar la aplicación como bibliotecas de carga dinámica.
Explicación	Basado en el sistema de trazas de Ogre3D todos los eventos relevantes en la aplicación son reportados en el fichero Ogre.log. Este sistema permite conocer dónde están los posibles fallos en la aplicación.

Tabla 8. Escenario 2.

Escenario #3	**El sistema es capaz de volver al último estado estable de la aplicación después de ocurrir un error.**
Atributo(s)	Confiabilidad – Recuperabilidad.
Estímulo	Se aplica un cierre forzado a la aplicación.
Respuesta	Al iniciar la aplicación, después del cierre forzado, debe responder de manera normal y no ser capaz de recuperar el último estado en que se encontraba antes del cierre.
Explicación	En la arquitectura no existe un mecanismo para recuperar estados ante la ocurrencia de errores fatales. Esto trae como consecuencia que el usuario pueda perder los datos sobre el ejercicio realizado a causa de un error de este tipo.

Tabla 9. Escenario 3.

Escenario #4	**El sistema debe permitir, después de desplegado, adicionar nuevos ejercicios sin muchas complicaciones.**

Atributo(s)	Mantenibilidad – Facilidad de adaptación al cambio.
Estímulo	Después de crear un laboratorio virtual con un ejercicio de prueba, se tratará de incorporar otro ejercicio con características diferentes.
Respuesta	La aplicación debe ser capaz de adicionar el nuevo ejercicio y permitir que el usuario interactúe con él sin complicaciones.
Explicación	Como parte de las especificaciones de la arquitectura los ejercicios son manejados en forma de componentes dinámicos independientes a la aplicación. Esto permite disminuir los costos de actualización de la aplicación e incorporar nuevos ejercicios de manera simple.

Tabla 10. Escenario 4.

Escenario #5	**Mantenimiento de algún componente.**
Atributo(s)	Mantenibilidad – Facilidad de adaptación al cambio.
Estímulo	Se cambia la implementación de un componente.
Respuesta	La aplicación debe ser capaz de trabajar con la nueva implementación del componente.
Explicación	Siempre y cuando no cambie la interfaz del componente la arquitectura podrá manejar la nueva implementación. Como ventaja esto permite tener un mismo componente con diferentes comportamientos para un uso personalizado.

Tabla 11. Escenario 5.

Escenario #6	**El sistema puede ser portado en varios sistemas operativos.**
Atributo(s)	Portabilidad – Adaptabilidad.

Estímulo	Se ejecuta la aplicación en el sistema operativo *Debian*.
Respuesta	La aplicación funciona correctamente.
Explicación	La arquitectura tiene como base de implementación el *framework* QT y el motor gráfico Ogre3D, ambos multiplataforma, lo que permite que a medida que estos sistemas sean portados a otras plataformas la arquitectura pueda ser portada de igual manera sin contratiempos.

Tabla 12. Escenario 6.

3.6.4 Resultados de la evaluación

Basada en la información recolectada y haciendo uso del método ATAM se pudo identificar 2 riesgos en los escenarios 1 y 3. Los riegos encontrados, después de analizados con el arquitecto principal y algunos desarrolladores, demuestran que la arquitectura puede presentar problemas en caso de que estos escenarios ocurran, pero en correspondencia con el tipo de aplicación a desarrollar, los usuarios que la utilizarán y la prioridad de estos escenarios se decide registrar y posponer el tratamiento de estos riesgos para una segunda iteración de la arquitectura.

Partiendo de la idea que la evaluación solo proporciona el potencial de arquitectura para alcanzar los atributos de calidad requeridos, se realizó una prueba de la arquitectura con los desarrolladores del proyecto Laboratorios Virtuales. Como objetivo principal de esta prueba se consideró:

- Probar la funcionalidad del diseño propuesto.
- Encontrar errores y funcionamientos no adecuados en la arquitectura.
- Dar a conocer al equipo de desarrollo la arquitectura candidata a utilizar para la creación de laboratorios virtuales.

Las pruebas realizadas arrojaron la presencia de errores en varios componentes de la arquitectura. También se constató que la familiarización con la arquitectura y su estructura general puede ser compleja. Se demostró que la arquitectura es completamente funcional y capaz de adaptarse aunque no se encuentre completamente terminada.

CONCLUSIONES

El diseño de la arquitectura de software es una de las etapas fundamentales en el proceso de desarrollo de software. En esta etapa son importantes los conocimientos y la experiencia adquirida por los arquitectos para obtener una buena solución. Al finalizar este trabajo, se arriban a las siguientes conclusiones:

- La arquitectura propuesta con un enfoque multiplataforma y basado en tecnologías libres representa una guía para el desarrollo de los laboratorios virtuales en el CEDIN.
- La combinación de los estilos y patrones de diseño utilizados permite alcanzar altos niveles de reutilización y mantenibilidad.
- Las pruebas realizadas, mediante el método ATAM, arrojaron la presencia de riesgos en la arquitectura los cuales fueron registrados.

RECOMENDACIONES

De forma general, el diseño arquitectónico propuesto describe toda una serie de aspectos significativos referentes al desarrollo de laboratorios virtuales. No obstante, como parte de un proceso de mejoras se proponen las siguientes recomendaciones:

- Incrementar la documentación asociada a la arquitectura de software.
- Se propone extender la funcionalidad de la arquitectura dándole soporte para el trabajo con gráficos en 2d.
- Mitigar los riesgos detectados al aplicar el método ATAM.

BIBLIOGRAFÍA

1. **Kruchten, Philippe.** *The "4+1" View Model of Software Architecture.* 1995.

2. *A Survey of Architecture Description Languages.* **Clements, Paul.** Alemania : s.n., 1996. IWSSD '96 Proceedings of the 8th International Workshop on Software Specification and Design. p. 16. ISBN: 0-8186-7361-3.

3. **IEEE Computer Society.** 1471-2000 - Recommended Practice for Architectural Description for Software-Intensive Systems. 2000.

4. **Jacobson, Ivar, Booch, Grady and Rumbaugh, James.** *El Proceso Unificado de Desarrollo de Software.* Madrid : Addison-Wesley, 2000. ISBN: 84-7829-036-2.

5. **Bass, Len, Clements, Paul and Kazman, Rick.** *Software architecture in practice.* s.l. : Addison-Wesley Professional, 2003. ISBN: 03-2115-4959.

6. *An introduction to software architecture.* **Garlan, David and Shaw, Mary.** 1994, Technical report (Carnegie Mellon University. School of Computer Science).

7. **Shaw, Mary and Garlan, David.** *Software Architecture: Perspectives on an emerging discipline.* Upper Saddle River : Prentice Hall, 1996.

8. *Engineering Context (for Software Architecture).* **Boehm, Barry.** Seattle : s.n., 1995. First International Workshop on Architecture for Software Systems.

9. **Reynoso, Carlos.** *Carlos Reynoso Investigación, Publicaciones y Cursos de Antropología, Ciencia Cognitiva y Complejidad.* [Online] 2004. carlosreynoso.com.ar/arquitectura-de-software/.

10. *Coming attractions in Software Architecture.* **Clements, Paul.** s.l. : Technical Report, 1996, Vols. CMU/SEI-96-TR-008, ESC-TR-96-008.

11. *The coming-of-age of Software Architecture research.* **Shaw, Mary.** 2001. Proceedings of the 23rd International Conference on Software Engineering.

12. *Software Architecture.* **Kazman, Rick.** [ed.] S-K Chang. s.l. : World Scientific Publishing, 2001, Handbook of Software Engineering and Knowledge Engineering.

13. *Foundations for the study of software.* **E. Perry, Dewayne and L. Wolf, Alexander.** 1992 : s.n., ACM SIGSOFT Software Engineering Notes.

14. *A field guide to Boxology: Preliminary classification of architectural styles for software systems.* **Shaw, Mary and Clements, Paul.** 1997. Proceedings of the 21st International Computer Software and Applications Conference.

15. *Attribute-based architectural styles.* **Klein, Mark and Kazman, Rick.** s.l. : Carnegie Mellon University, 1999. CMU/SEI-99-TR-022.

16. *The Wright Architectural Description Language.* **Allen, Robert and Garlan, David.** s.l. : Carnegie Mellon University, 1996.

17. *Architectural styles and the design of network-based.* **Fielding, Roy Thomas.** Irvine : University of California, 2000. Tesis doctoral.

18. *Microsoft.* [Online] http://msdn.microsoft.com/en-us/library/ms978678.

19. *Microsoft.* [Online] http://geeks.ms/blogs/jkpelaez/archive/2009/04/18/arquitectura-basada-en-componentes.aspx.

20. **Szyperski, Clemens Alden.** *Component software: Beyond Object-Oriented programming.* s.l. : Addison-Wesley, 2002.

21. *Desarrollo de Software Basado en Componentes.* **Fuentes, Lidia, Troya, Jose and Vallecillo, Antonio.** España : Universidad de Malaga.

22. *A Pattern Language.* **Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, Shlomo Angel.** New York : Oxford University Press, 1977.

23. **Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerlad, Peter; Stal, Michael.** *Pattern-oriented software architecture – A system of patterns.* 1996 : John Wiley & Sons.

24. **Gamma, Erich; Helm, Richard ; Johnson, Ralph; Vlissides, John.** *Design Patterns: Elements of reusable object-oriented software.* s.l. : Addison-Wesley, 1995.

25. **Larman, Craig.** *Applying UML and Patterns An Introduction to Object-Oriented Analysis an Design and the Unified Process.* s.l. : Prentice Hall.

26. *Informe de la reunión de expertos sobre laboratorios virtuales.* **Vary, James P.** París : Organización de las Naciones Unidas para la Educación, la Ciencia y la Cultura, 1999.

27. *Ventajas y Desventajas de usar Laboratorios Virtuales en educacion a distancia: la opinión del estudiantado en un proyecto de seis años de duración.* **Monge Nájera, Julián**

and Hugo Méndez, Víctor. 1, Costa Rica : Universidad de Costa Rica, 2007, Vol. 31. ISSN: 0379-7082.

28. *Laboratorios remotos y virtuales en enseñanzas técnicas y científicas.* **Calvo, Isidro.** s.l. : UPV/EHU.

29. Universidad de Oxford. *Virtual Experiments.* [Online] http://www.chem.ox.ac.uk/vrchemistry/labintro/newdefault.html.

30. UNED. *AutomatL@bs.* [Online] http://lab.dia.uned.es/automatlab.

31. Università degli Studi di Siena. *Automatic Control Telalab.* [Online] http://act.dii.unisi.it/home.php.

32. Universidad de León. *Laboratorio Remoto de Automática.* [Online] http://lra.unileon.es/es.

33. *Sibees Soft .* [Online] http://www.sibees.com.

34. *Design Simulation Technologies, Inc.* [Online] http://www.design-simulation.com/ip/index.php.

35. **C. Michael Pilato, Ben Collins-Sussman, Brian W. Fitzpatrick.** *Version Control with Subversion.* s.l. : O'Reilly Media, 2008. 978-0596510336.

36. **Allison, Chuck.** *Fundamentos de Java y C + +.* s.l. : MindView, 2000.

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

38. **Bosch, Jan.** *Design and use of software architectures: adopting and evolving a product-line approach.* s.l. : Addison-Wesley, 2000. ISBN 9780201674941.

39. *Quality Characteristics for Software Architecture.* **Francisca Losavio, Ledis Chirinos, Nicole Lévy, Amar Ramdane-Cherif.** s.l. : Journal of Object Technology 2, 2003.

40. *Software Architecture: A Roadmap.* **Garlan, David.** [ed.] Anthony Finkelstein. s.l. : ACM Press, 2000, The future of software engineering.

41. *Software architecture: An executive overview.* **Clements, Paul and Northrop, Linda.** 1996.

42. **Burbeck, Steve.** Application programming in Smalltalk-80: How to use Model-View-Controller (MVC)". [Online] 1992. http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html.

43. *Microsoft.* [Online] http://msdn.microsoft.com/en-us/library/ff649643.aspx.

44. *Quality Attributes.* **Barbacci, Klein, Longstaff, Weinstock.** s.l. : Carnegie Mellon University, 1995.

45. *Patrones de Alexander a la Tecnología de Objetos.* **Peñalvo, Francisco García.** Salamanca : s.n., 1998, Revista Profesional para Programadores.

GLOSARIO DE TÉRMINOS.

Framework: en el desarrollo de *software*, es una estructura de soporte en la que otro proyecto de software 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.

Render: En Gráfico por Computadoras denota el proceso de sintetizado de imágenes, es decir, la traducción de la geometría controlada por atributos gráficos, al medio de la imagen. De una forma más general renderizar incluye la interpretación y la evaluación de componentes gráficos para un medio en específico.