Universidad de las Ciencias Informáticas

Facultad 6

Título: Propuesta del diseño arquitectónico de la Plataforma bioGRATO

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

Autores

Yadira Marrero López

Adonis Ricardo Rosales García

Tutores

MSc. Aurelio Antelo Collado

Ing. Yuleydis Mejias Cesar

Lic. Rafael Arturo Trujillo Rasúa

Junio de 2008

Declaración de Autoría

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

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

Yadira Marrero López

_____

Firma del Autor

Adonis Ricardo Rosales García

_____

Firma del Autor

MSc. Aurelio Antelo Collado

_____

Firma del Tutor

Ing. Yuleydis Mejias Cesar

_____

Firma del Tutor

Lic. Rafael Arturo Trujillo Rasúa

_____

Firma del Tutor

"Lo que sabemos es una gota de agua; lo que ignoramos es el océano."

Isaac Newton

Datos de contacto

Tutores:

Aurelio Antelo Collado: Ingeniero Industrial, Master en Matemática Aplicada, Profesor Asistente con 3 años de experiencia.
Email: aantelo@uci.cu

Yuleydis Mejias Cesar: Ingeniera en Ciencias Informáticas.
Email: ymejias@uci.cu

Rafael Arturo Trujillo: Licenciado en Ciencias de la Computación, Profesor Instructor con 5 años de experiencia.
Email: Trujillo@uci.cu

Agradecimientos

Yadira:

Muy en especial a mis padres, por su esfuerzo, sacrificio y amor de toda la vida.

A Nancy y Abundio por ser mis segundos padres.

A mi hermana y sobrina por ser tan especiales.

A Ricky por ser mi amor, amigo y consejero.

A toda mi familia, por su apoyo incondicional.

A los profes Trujillo y Yoisell por su ayuda en todo momento.

A todos mis amigos, en especial a Leslier, Vladimir y Elsia.

Adonis:

A mis padres, por su amor y cariño infinitos, por su entrega y sacrificio, por confiar en mí.

A Yasirys, mi gran amor, por su cariño y compañía, por su entrega incondicional.

A mi abuelo Elpidio, donde quiera que esté, por enseñarme a ser responsable, por encaminarme en la vida.

A mi hermano Omar, por su ayuda y ejemplo.

A Yamirka y a Guido, por su ayuda, por convertirse en mi familia.

A Ángela, por su apoyo y cariño.

A mi familia en general.

A mis amigos y compañeros de aula, por el tiempo compartido.

Ambos:

A la Revolución por brindarnos la posibilidad de estudiar en esta maravillosa escuela.

A Fidel por ser el creador de este proyecto.

A nuestros tutores por la ayuda brindada.

Dedicatoria

A nuestros padres, artífices principales de nuestras vidas.

Resumen

La investigación surge en el marco de trabajo del proyecto desarrollado de forma conjunta por el Centro de Química Farmacéutica (CQF) y la Universidad de las Ciencias Informáticas (UCI). Dada la necesidad de integrar los módulos de la Plataforma Inteligente para la Predicción de Actividad Biológica de Compuestos Orgánicos (bioGRATO), se decidió diseñar una arquitectura que permita esta integración. La arquitectura diseñada proporciona los elementos necesarios para el desarrollo de una plataforma que brinde a los especialistas químicos un entorno de trabajo provisto de las funcionalidades necesarias para la evolución de sus investigaciones. Su éxito viene dado por las decisiones arquitectónicas que fueron tomadas durante su desarrollo, por la definición de herramientas, tecnologías y aspectos esenciales de diseño. La arquitectura está basada en la concepción del uso de herramientas libres. Se hizo una evaluación de la propuesta, obteniendo resultados satisfactorios.

Palabras clave

Arquitectura de software

Índice

Introducción

El medicamento es un valor social, y la industria farmacéutica es la principal fuente de esperanza ante nuevos retos que le obligan a un constante esfuerzo en la búsqueda de nuevos fármacos para el tratamiento de enfermedades. El Centro de Química Farmacéutica, fue creado en Cuba con la misión fundamental de llevar a cabo el desarrollo de investigaciones científico-técnicas dirigidas a la obtención de sustancias bioactivas para la formulación de medicamentos de uso humano.

En la actualidad, se ha tornado una necesidad en la industria químico-farmacéutica el desarrollo de aplicaciones informáticas para extraer conocimiento de la información generada en los laboratorios, actividad que está basada en la búsqueda en grandes bases de datos.

En el mundo existen aplicaciones para visualizar y editar estructuras químicas, como por ejemplo HyperChem y CORINA, otras como el Codessa y Dragon, permiten el cálculo de descriptores. Estas herramientas aunque poseen múltiples funcionalidades, presentan algunos inconvenientes, por ejemplo HyperChem y CORINA no tienen en cuenta la predicción de actividades biológicas o el cálculo de descriptores; el Codessa y Dragon carecen de interfaz de visualización y edición molecular, siendo esto una deficiencia, pues entorpece el desempeño de los usuarios. Todas estas solo se centran en objetivos específicos para los cuales fueron diseñadas. Además, algunas son muy costosas e inaccesibles para Cuba.

El país avanza aceleradamente en el desarrollo de la Bioinformática. En esta tarea la Universidad de las Ciencias Informáticas, cumpliendo con una de las misiones para las que fue creada: producir software y servicios informáticos, a partir de la vinculación estudio-trabajo como modelo de formación, desempeña un rol fundamental en su colaboración con el CQF en la creación de varios proyectos informáticos. La facultad 6 participa directamente en el desarrollo de estos proyectos entre los que se encuentra: “Plataforma Inteligente para la Predicción de Actividad Biológica de Compuestos Orgánicos”.

En la actualidad, bioGRATO cuenta con un conjunto de módulos que permiten trabajar con moléculas y proteínas, empleando técnicas de Inteligencia Artificial para la predicción de actividad biológica y modelos desarrollados a partir de una base de datos de compuestos con actividad reportada. Cada módulo trabaja de forma independiente, sin embargo, en múltiples ocasiones se hace necesaria la comunicación con otros módulos, para en base a la información obtenida de ellos, completar la solución deseada. Para los investigadores que utilizan cada una de las aplicaciones esta situación

representa una limitante significativa, pues conlleva a gastos innecesarios de esfuerzo, tiempo y dinero.

En un primer intento se trató de solucionar la incomunicación entre los módulos mediante el uso de plug-in, pero esto sólo funcionaba en un entorno local, lo cual es ineficiente ya que cualquier usuario autorizado a trabajar con la herramienta, y que no se encuentre en el laboratorio de trabajo, no podría favorecerse de la misma. Otra dificultad considerable es que los resultados alcanzados por los grupos de investigación ubicados en espacios geográficos distantes, no se encontrarían accesibles en tiempo real para los restantes grupos porque estarían almacenando la información en Bases de Datos (BD) diferentes.

Teniendo en cuenta lo analizado se plantea como **Problema Científico** de la investigación: ¿Cómo lograr la integración entre los módulos de la Plataforma bioGRATO?

En correspondencia con el problema científico planteado se define como **objeto de estudio** la Arquitectura de Software (AS), cuyo **campo de acción** está enmarcado en la Arquitectura de Software para aplicaciones de Predicción de Actividad Biológica de Compuestos Orgánicos.

El **objetivo general** de la investigación es diseñar la arquitectura de la plataforma bioGRATO. La consecución de este estará sustentada en los **objetivos específicos** siguientes:

1. Seleccionar estilos y patrones de arquitectura.
2. Diseñar las vistas del sistema.
3. Describir la arquitectura del sistema.
4. Evaluar el diseño arquitectónico propuesto.

Con el fin de dar cumplimiento a los objetivos de esta investigación, se plantean las siguientes **tareas** de la investigación:

1. Revisión de los estilos arquitectónicos existentes.
2. Revisión de los patrones de arquitectura existentes.
3. Identificación de los estilos arquitectónicos a utilizar, fundamentación de estos estilos, así como su aplicación.
4. Identificación de los patrones arquitectónicos a utilizar, fundamentación de estos patrones, así como su aplicación.

5. Definición y fundamentación de las herramientas a utilizar para el desarrollo de la plataforma.
6. Análisis y diseño de las vistas del sistema.
7. Definición de las estrategias de comunicación e integración de los módulos.
8. Representación del diseño arquitectónico propuesto.
9. Selección y aplicación de un método para validar el diseño arquitectónico propuesto.

El presente documento está estructurado en tres capítulos.

Capítulo 1: Fundamentación Teórica.

En este capítulo se presentan conceptos asociados al objeto de estudio, así como los diferentes tipos de arquitectura y patrones arquitectónicos más usados mundialmente. Se aborda acerca de los lenguajes, tecnologías y herramientas existentes; haciendo una selección de aquellos que son usados para dar solución al problema científico.

Capítulo 2: Descripción de la arquitectura.

En este capítulo se hace una propuesta de solución al problema planteado. Se diseñan las vistas arquitectónicas y se hace una descripción de la arquitectura para la plataforma, definiendo cómo debe hacerse la integración y comunicación entre los módulos.

Capítulo 3: Evaluación del diseño arquitectónico propuesto.

En este capítulo se presentan los distintos métodos para evaluar un diseño arquitectónico y a partir del método seleccionado se hace un análisis de la solución propuesta.

Capítulo

Fundamentación teórica

1.1 Introducción

En el presente capítulo se plantean los principales conceptos que se relacionan con el objeto de estudio y que son necesarios para entender posteriormente la propuesta de solución. Se estudian puntos medulares de la Arquitectura de Software, como son: el análisis de los estilos arquitectónicos y los patrones de arquitectura más utilizados. Se analizan los Lenguajes de Descripción Arquitectónica (ADLs), tecnologías, lenguajes de programación y herramientas a utilizar para el desarrollo de la plataforma.

1.2 Arquitectura de Software

Cada vez que se narra la historia de la arquitectura de software (o de la ingeniería de software, según el caso), se reconoce que en un principio, hacia 1968, Edsger Dijkstra, de La Universidad Tecnológica de Eindhoven en Holanda y Premio Turing 1972, propuso que se establezca una estructuración correcta de los sistemas de software antes de lanzarse a programar, escribiendo código de cualquier manera. Dijkstra, sostenía que las ciencias de la computación eran una rama aplicada de las matemáticas y sugería seguir pasos formales para descomponer problemas mayores; aunque Dijkstra no utiliza el término arquitectura para describir el diseño conceptual del software, sus conceptos sientan las bases. [1]

En 1975, Brooks, diseñador del sistema operativo OS/360 y Premio Turing 2000, identificaba y razonaba sobre las estructuras de alto nivel y reconocía la importancia de las decisiones tomadas a ese nivel de diseño. También distinguía entre arquitectura e implementación; mientras la primera decía qué hacer, la implementación se ocupa de cómo. En la década de 1980, los métodos de desarrollo estructurado demostraron no escalar suficientemente y fueron dejando el lugar a un nuevo paradigma, el de la programación orientada a objetos (POO). Paralelamente, hacia fines de la década de 1980 y comienzos de la siguiente, la expresión arquitectura de software comienza a aparecer en la literatura para hacer referencia a la configuración morfológica de una aplicación. [1]

El primer estudio en que aparece la expresión “arquitectura de software” en el sentido en que hoy lo conocemos es sin duda el de Perry y Wolf; ocurrió tan tarde como en 1992, aunque el trabajo se fue gestando desde 1989. En él, los autores proponen concebir la AS por analogía con la arquitectura de edificios, una analogía de la que luego algunos abusaron, otros encontraron útil y para unos pocos ha devenido inaceptable.[2]

Dando cumplimiento a las profecías de Perry y Wolf, la década de 1990 fue sin duda la de la consolidación y diseminación de la AS en una escala sin precedentes. Las contribuciones más importantes surgieron en torno del instituto de ingeniería de la información de la Universidad Carnegie Mellon. En la misma década, demasiado pródiga en acontecimientos, surge también la programación basada en componentes, que en su momento de mayor impacto impulsó a algunos arquitectos a afirmar que la AS promovía un modelo que debía ser más de integración de componentes preprogramados que de programación.[3]

Un segundo gran tema de la época fue el surgimiento de los patrones, cristalizada en dos textos fundamentales, el de la Banda de los Cuatro en 1995 [4] y la serie POSA (del inglés *Pattern-Oriented Software Architecture*) desde 1996. El primero de ellos promueve una expansión de la programación orientada a objetos, mientras que el segundo desenvuelve un marco más ligado a la AS.

La AS de este período realizó su trabajo de homogenización de la terminología, desarrolló la tipificación de los estilos arquitectónicos y elaboró lenguajes de descripción de arquitectura.

En el siglo XXI, la AS aparece dominada por estrategias orientadas a líneas de productos y por establecer modalidades de análisis, diseño, verificación, refinamiento, recuperación, diseño basado en escenarios, estudios de casos y hasta justificación económica, redefiniendo todas las metodologías ligadas al ciclo de vida en términos arquitectónicos. Todo lo que se ha hecho en ingeniería debe formularse de nuevo, integrando la AS en el conjunto.

1.2.1 Definición de Arquitectura de Software

La arquitectura del software nos proporciona una visión global del sistema a construir. Describe la estructura y la organización de los componentes del software, sus propiedades y las conexiones entre ellos. Los componentes del software incluyen módulos de programas y varias representaciones de datos que son manipulados por el programa. Además, el diseño de datos es una parte integral para la derivación de la arquitectura del software. La arquitectura marca decisiones de diseño tempranas y proporciona el mecanismo para evaluar los beneficios de las estructuras de sistema alternativas.[5]

Se han propuesto muchas definiciones de arquitectura de software, a continuación se hace referencia a las que se consideran las más acertadas.

Según Len Bass, Paul Clements y Rick Kazman:

“La arquitectura de software de un sistema de programa o computación es la estructura de las estructuras del sistema, la cual comprende los componentes del software, las propiedades de esos componentes visibles externamente, y las relaciones entre ellos.”[6]

Philippe Kruchten plantea:

“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.”[7]

La definición que se ha acordado como oficial es la provista por el documento de lEEE STD 1471-2000, que expresa:

“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”.

Ninguna definición de la AS es respaldada unánimemente por la totalidad de los arquitectos; independientemente de las discrepancias entre las diversas definiciones, es común entre todos los autores el concepto de la arquitectura como un punto de vista que concierne a un alto nivel de abstracción. El concepto de abstracción es tomado en el sentido de extraer las propiedades esenciales o examinar selectivamente ciertos aspectos de un problema, posponiendo o ignorando los detalles menos sustanciales o irrelevantes.

La arquitectura no es el software operacional. Es la representación que capacita al ingeniero del software para: (1) analizar la efectividad del diseño para la consecución de los requisitos fijados, (2) considerar las alternativas arquitectónicas en una etapa en la cual hacer cambios en el diseño es relativamente fácil, y (3) reducir los riesgos asociados a la construcción del software.[5]

1.3 Estilos y patrones

En algunas bibliografías referentes al estudio de los patrones arquitectónicos se suele llamar de la misma manera a los patrones y estilos, pero la mayoría de los autores los ven de manera separada. Ambos se refieren a formas de estructurar los sistemas y cómo relacionar los componentes de estos, la diferencia radica en el nivel de abstracción en que se aplican. Los estilos están en un plano de abstracción más alto, definiendo como configurar la arquitectura, mientras los patrones están más cercanos al diseño, incluso podría decirse que más cercano a algo físico, pues estos pueden representarse mediante código en un lenguaje de programación determinado.

Vale la pena aclarar la relación entre estilos, patrones de diseño y patrones de arquitectura. Los patrones de diseño de software buscan codificar y hacer reutilizables un conjunto de principios a fin de diseñar aplicaciones de alta calidad. Se aplican en principio solo en la fase de diseño, aunque la comunidad ha comenzado a definir y aplicar patrones a otras etapas del proceso de desarrollo. Los estilos se han aplicado en la fase de análisis arquitectónico en términos de patrones de arquitectura. Sin bien los patrones de arquitectura son similares a los de diseño, los primeros se concentran en la estructura de alto nivel del sistema.

Algunos patrones coinciden con los estilos hasta en el nombre conque se los designa. Los elementos por los que difieren son mostrados la tabla 1.[8]

Tabla 1. Diferencias entre Estilos y Patrones Arquitectónicos.

Elemento a comparar	**Estilo Arquitectónico**	**Patrón Arquitectónico**
Definición	Son una categorización de sistemas.	Son soluciones generales a problemas comunes.
Representación	Sólo describe el esqueleto estructural general para aplicaciones.	Existen en varios rangos de escala, comenzando con patrones que definen la estructura básica de una aplicación.
Dependencia de contexto	Son independientes del contexto al que puedan ser aplicados.	Partiendo de la definición de patrón, requieren de la especificación de un contexto del problema.

Solución	Expresan técnicas de diseño desde una perspectiva que es independiente de la situación actual de diseño.	Expresa un problema recurrente de diseño muy específico, y presenta una solución para él, desde el punto de vista del contexto en el que se presenta.

Una vez analizados los elementos esenciales de los estilos y patrones de arquitectura, se concluye que si bien existen convergencias entre ambos conceptos, los patrones se refieren más a prácticas de reutilización y los estilos conciernen a teorías sobre la estructura de los sistemas.

1.3.1 Estilos

Los estilos arquitectónicos de software son arquitecturas de software comunes, marcos de referencias arquitectónicas, formas comunes o clases de sistemas.

Se entiende por estilos a las entidades que ocurren en un nivel sumamente abstracto, puramente arquitectónico, que no coincide ni con la fase de análisis propuesta por la temprana metodología de modelado orientada a objetos (aunque sí un poco con la de diseño), ni con lo que más tarde se definirían como paradigmas de arquitectura, ni con los patrones arquitectónicos.[9]

A la hora de definir un estilo arquitectónico es necesario tener en cuenta el tipo de aplicación, ya que puede imponer restricciones que acotan la interacción de los componentes, además se tiene en cuenta el patrón de organización general.

Algunos de los principales estilos arquitectónicos que se usan en la actualidad están divididos por Clases de Estilos las que se exponen a continuación:[9]

- **Estilos de flujo de datos:** Esta familia de estilos enfatiza la reutilización y la modificabilidad. Es apropiada para sistemas que implementan transformaciones de datos en pasos sucesivos.
    - **Arquitecturas en tubería y filtros:** El sistema tubería y filtros se percibe como una serie de transformaciones sobre sucesivas piezas de los datos de entrada. Los datos entran al sistema y fluyen a través de los componentes. El supuesto es que cada paso se ejecuta hasta completarse antes que se inicie el paso siguiente.

      Ventajas: Es simple de entender e implementar. Es posible implementar procesos complejos con editores gráficos de líneas de tuberías o con comandos de línea. Fuerza un

procesamiento secuencial. Los filtros se pueden empaquetar, y hacer paralelos o distribuidos.

Desventajas: Eventualmente pueden llegar a requerirse *buffers* de tamaño indefinido, por ejemplo en las tuberías de clasificación de datos. El estilo no es apto para manejar situaciones interactivas, sobre todo cuando se requieren actualizaciones incrementales de la representación en pantalla. La independencia de los filtros implica que es muy posible la duplicación de funciones de preparación que son efectuadas por otros filtros.

- **Estilos centrados en datos:** Esta familia de estilos enfatiza la integrabilidad de los datos. Se estima apropiada para sistemas que se fundan en acceso y actualización de datos en estructuras de almacenamiento.

    - **Arquitecturas de pizarra o repositorio:** En esta arquitectura hay dos componentes principales: una estructura de datos que representa el estado actual y una colección de componentes independientes que operan sobre él.

      Ventajas: Hace posible la interacción de agentes contra el sistema. Funciona muy bien con los problemas no deterministas. Se sabe el conocimiento que se tiene en cada momento del proceso.

      Desventajas: Problemas de seguridad ya que la pizarra es accesible por todos. Problemas de sincronización al chequear/vigilar la pizarra.

- **Estilos de llamada y retorno:** Esta familia de estilos enfatiza la modificabilidad y la escalabilidad. Son los estilos más generalizados en sistemas en gran escala. El sistema se constituye de un programa principal que controla el sistema y varios subprogramas que se comunican con éste mediante el uso de llamadas.

    - **Modelo Vista Controlador (MVC):** El estilo conocido como Modelo Vista Controlador (MVC) separa el modelado del dominio, la presentación y las acciones basadas en datos ingresados por el usuario en tres clases diferentes: Modelo, Vista y Controlador.

      Ventajas: Soporte de vistas múltiples. Dado que la vista se halla separada del modelo y no hay dependencia directa del modelo con respecto a vista, la interfaz de usuario puede mostrar múltiples vistas de los mismos datos simultáneamente. Adaptación al cambio.

Desventajas: Costo de actualizaciones frecuentes. Desacoplar el modelo de la vista no significa que los desarrolladores del modelo puedan ignorar la naturaleza de las vistas. Incrementa la complejidad de la solución.

- **Arquitecturas en capas:** Definida por Garlan y Shaw 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.

  Ventajas: Soporta un diseño basado en niveles de abstracción crecientes. Admite muy naturalmente optimizaciones y refinamientos; proporciona amplia reutilización.

  Desventajas: Muchos problemas no admiten un buen mapeo en una estructura jerárquica. A veces es también extremadamente difícil encontrar el nivel de abstracción correcto. Además, los cambios en las capas de bajo nivel tienden a filtrarse hacia las de alto nivel.

- **Arquitecturas orientadas a objetos**: Los componentes del estilo se basan en principios orientados a objetos como encapsulamiento, herencia y polimorfismo. Entre las cualidades de este estilo, la más básica concierne a que se puede modificar la implementación de un objeto sin afectar a sus clientes.

  Ventaja: Permite la estandarización de interfaces de componentes.

  Desventajas: Los componentes se encuentran fuertemente acoplados. Sólo soportan interacciones atómicas.

- **Arquitecturas basadas en componentes:** Los sistemas de software basados en componentes se basan en principios definidos por una ingeniería de software específica. Los componentes son las unidades de modelado, diseño e implementación. Las interfaces están separadas de las implementaciones, y las interfaces y sus interacciones son el centro de incumbencias en el diseño arquitectónico. Los componentes soportan algún régimen de introspección, de modo que su funcionalidad y propiedades puedan ser descubiertas y utilizadas en tiempo de ejecución.

  Ventajas: Permite alcanzar un mayor nivel de reutilización de software. Permite que las pruebas sean ejecutadas probando cada uno de los componentes antes de probar el conjunto completo de componentes ensamblados. Simplifica el mantenimiento del sistema, cuando existe un débil acoplamiento entre componentes, el desarrollador es libre de actualizar y/o agregar componentes según sea necesario, sin afectar otras partes del

sistema. Dado que un componente puede ser construido y luego mejorado continuamente por un experto u organización, la calidad de una aplicación basada en componentes mejorará con el paso del tiempo.

Desventajas: Las actualizaciones de los componentes adquiridos no están en manos de los desarrolladores del sistema. Si no existen los componentes, toca desarrollarlos y se puede perder mucho tiempo, así como que estos componentes pueden tener conflictos si de estos sale una nueva versión y no está estandarizado con lo que se ha desarrollado en la aplicación ensamblada.

- **Estilos de Código Móvil:** Esta familia de estilos enfatiza la portabilidad. Ejemplos de la misma son los intérpretes, los sistemas basados en reglas y los procesadores de lenguaje de comando.

    - **Arquitecturas de Máquinas Virtuales (MV):** Se ha llamado también intérpretes basados en tablas. Comprende básicamente dos formas o subestilos, que se han llamado intérpretes y sistemas basados en reglas.

      Ventajas: La principal ventaja es la portabilidad del código. La generación de una aplicación requiere el conocimiento concreto de la plataforma en la que vamos a ejecutar el código, pero sin embargo al utilizar una MV este problema desaparece.

      Desventajas: No son tan rápidos como los lenguajes compilados, pero si son bastante más rápidos que los interpretados, ya que aún teniendo que codificar las instrucciones, esta es mucho más sencilla. Se pierde claridad, ya que no se tiene el código fuente. Del mismo modo, para encontrar un error es más complejo.

- **Estilos heterogéneos:** En esta familia se clasifican aquellos sistemas que no pueden encajar exactamente en ninguno de los tipos anteriores.

    - **Sistemas de control de procesos:** Desde el punto de vista arquitectónico, mientras casi todos los demás estilos se pueden definir en función de componentes y conectores, los sistemas de control de procesos se caracterizan no sólo por los tipos de componentes, sino por las relaciones que mantienen entre ellos. El objetivo de un sistema de esta clase es mantener ciertos valores dentro de ciertos rangos especificados, llamados puntos fijos o valores de calibración.

    - **Arquitecturas Basadas en Atributos**: La arquitectura basada en atributos fue propuesta para asociar a la definición del estilo arquitectónico un *framework* de razonamiento (ya sea

cuantitativo o cualitativo), basado en modelos de atributos específicos. Su objetivo se funda en la premisa que dicha asociación proporciona las bases para crear una disciplina de diseño arquitectónico, tornando el diseño en un proceso predecible.

- **Estilos *Peer-to-Peer*:** Esta familia se conoce también como componentes independientes, enfatiza la modificabilidad por medio de la separación de las diversas partes que intervienen en la computación. Consiste por lo general en procesos independientes o entidades que se comunican a través de mensajes.

    - **Arquitecturas basadas en eventos:** Se vinculan históricamente con sistemas basados en publicación-suscripción. Los conectores de estos sistemas incluyen procedimientos de llamada tradicionales y vínculos entre anuncios de eventos e invocación de procedimientos. La idea dominante en la invocación implícita es que, en lugar de invocar un procedimiento en forma directa, un componente puede anunciar mediante difusión uno o más eventos.

      Ventajas: Se optimiza el mantenimiento haciendo que procesos de negocios que no están relacionados sean independientes. Se alienta el desarrollo en paralelo, lo que puede resultar en mejoras de *performance*. Es fácil de empaquetar en una transacción atómica. Se puede agregar un componente registrándolo para los eventos del sistema; se pueden reemplazar componentes.

      Desventajas: El estilo no permite construir respuestas complejas a funciones de negocios. Un componente no puede utilizar los datos o el estado de otro componente para efectuar su tarea. Cuando un componente anuncia un evento, no tiene idea sobre qué otros componentes están interesados en él, ni el orden en que serán invocados, ni el momento en que finalizan lo que tienen que hacer.

    - **Arquitecturas orientadas a servicios (SOA):** Es lo suficientemente flexible, elegante y ágil garantizando las soluciones que las empresas han anhelado siempre. Es una arquitectura de software que construye toda la topología de la aplicación como una topología de interfaces, implementaciones y llamados a interfaces. Es una relación entre servicios y consumidores de servicios, ambos lo suficientemente amplios como para representar una función de negocio completa.

      Ventajas: Son varios los beneficios técnicos de una implementación de SOA entre ellos la repotenciación del software anterior, conectividad, facilidad de mantenimiento, reducción de

tamaño de proyectos, alta escalabilidad, reutilización real de los programas y mejora en tiempos de respuesta.

Desventajas: Dentro de los campos donde no se aconseja introducir SOA cabe mencionar aquellos donde frente a la flexibilidad se prefiera una centralización de la información, y ya se cuente con los programas y aplicaciones para ello. Otro campo donde SOA puede resultar menos atractiva es la relacionada con temas que requieran alta seguridad, resulta más complejo hacerlo con múltiples procesos independientes preparados fácilmente para compartir información que con un reducido número dentro de una aplicación monolítica.

- **Arquitecturas basadas en recursos:** Define recursos identificables y métodos para acceder y manipular el estado de esos recursos. El caso de referencia es nada menos que la *World Wide Web*, donde los URLs identifican los recursos y HTTP es el protocolo de acceso. El argumento central es que HTTP mismo, con su conjunto mínimo de métodos y su semántica simplísima, es suficientemente general para modelar cualquier dominio de aplicación.

1.3.2 Patrones de Software

Los patrones como elementos de reutilización tienen sus principios en la arquitectura (en el sentido clásico), donde fueron introducidos con el objetivo de capturar y posteriormente utilizar diseños que se habían aplicado en otras construcciones y que se catalogaron como complejos.

El arquitecto Christopher Alexander fue el primero en tratar de crear un formato específico para patrones en la arquitectura. Argumentaba que los métodos comunes aplicados en la disciplina daban lugar a productos que no satisfacían las demandas y requerimientos de los usuarios, y que eran ineficientes a la hora de conseguir el propósito de todo diseño y esfuerzo de la ingeniería: mejorar la condición humana.

Los patrones para el desarrollo de software constituyen un avance en la Tecnología Orientada a Objetos. Los mismos participan en la intención de capturar, recopilar, y transmitir la experiencia del diseñador.[10]

Las características deseables de un buen patrón son:

- Debe solucionar un problema: los patrones capturan soluciones, no solo principios o estrategias abstractas
- Debe ser un concepto probado: ser soluciones demostradas, no teorías o especulaciones.

- La solución no es obvia: muchas técnicas de solución de problemas tratan de hallar soluciones por medio de principios básicos. Los mejores patrones generan una solución a un problema de forma indirecta.
- Describe participantes y relaciones entre ellos: los patrones no sólo describen módulos sino estructuras del sistema y mecanismos más complejos.

Los elementos esenciales que debe contener un patrón son:[11]

- **Nombre:** Tiene que tener un nombre significativo. Sería muy incontrolable tener que describir el patrón cada vez que se utilizan en una discusión.
- **Problema:** Describe cuándo aplicarlo, explica el problema y su contexto y la lista de precondiciones que deben encontrarse, si las hubiera.
- **Solución:** Describe los elementos que lo componen: clases, objetos, relaciones, responsabilidades y colaboraciones.
- **Consecuencias:** Describe los costos y beneficios de aplicarlo. Incluye el impacto sobre la flexibilidad, extensibilidad y portabilidad del sistema

Los patrones pueden dividirse o clasificarse en:[10]

- **Patrones de arquitectura:** Relacionados a la interacción de objetos dentro o entre niveles arquitectónicos, problemas arquitectónicos, adaptabilidad a requerimientos cambiantes, modularidad, acoplamiento.
- **Patrones de diseño:** Fueron construidos dado los problemas con la claridad de diseño, multiplicación de clases, adaptabilidad a requerimientos cambiantes.
- **Patrones de análisis:** Usualmente específicos de aplicación.
- **Patrones de proceso o de organización:** Tratan todo lo concerniente con el desarrollo o procesos de administración de proyectos, o técnicas, o estructuras de organización.
- **Patrones de idioma:** Regulan la nomenclatura en la cual se escriben, se diseñan y desarrollan nuestros sistemas.

Una vez analizados los patrones de software y teniendo en cuenta el propósito del presente trabajo, resulta necesario analizar con mayor profundidad las clases: Patrones arquitectónicos y Patrones de diseño.

1.3.2.1 Patrones arquitectónicos

Los patrones de arquitectura expresan el esquema fundamental de organización para sistemas de software. Proveen un conjunto de subsistemas predefinidos; especifican sus responsabilidades e incluyen reglas y guías para organizar las relaciones entre ellos. Los patrones de arquitectura representan el nivel más alto en el sistema de patrones. Ayudan a especificar la estructura fundamental de una aplicación. Cada actividad de desarrollo es gobernada por esta estructura; por ejemplo, el diseño detallado de los subsistemas, la comunicación y colaboración entre diferentes partes del sistema. Cada patrón de arquitectura ayuda a conseguir una propiedad específica en el sistema global.[12]

Entre las ventajas del uso de patrones, se pueden encontrar:

- Permiten la reutilización de soluciones arquitectónicas de calidad.
- Son de gran ayuda para controlar la complejidad de un diseño.
- Facilitan la documentación de diseños arquitectónicos.
- Proporcionan un vocabulario común que mejora la comunicación entre diseñadores.

Una buena referencia en este tema lo constituye el libro "*Pattern-Oriented Software Architecture: A System of Patterns*" de F. Buschmann, donde los patrones propuestos son:

- **Tubería y filtros:** Provee una estructura para sistemas que procesan un flujo de datos. Cada etapa del proceso es encapsulada como un filtro. Los datos se pasan entre filtros adyacentes mediante tubos. Recombinando filtros se obtienen familias de sistemas relacionados.
- **Pizarra o repositorio:** Útil para sistemas en que no se conoce una solución o estrategia determinista. Varios subsistemas especializados ensamblan su conocimiento para construir una posible solución parcial.
- **Modelo Vista Controlador:** Divide una aplicación interactiva en tres componentes: el modelo contiene la información y funcionalidad principal, las vistas muestran información al usuario y el controlador gestiona la entrada de usuario. Un mecanismo de propagación de cambios asegura la consistencia entre el modelo y la interfaz de usuario.
- **Capas:** Permite estructurar aplicaciones que se pueden descomponer en grupos de subtareas, donde cada grupo está en un determinado nivel de abstracción.

- ***Broker:*** Permite estructurar sistemas distribuidos desacoplados que interaccionan mediante invocación de servicios remotos. Un componente *Broker* es responsable de coordinar la comunicación, así como de transmitir resultados y excepciones.
- ***Presentation Abstraction Control:*** Estructura una aplicación interactiva como una jerarquía de agentes que cooperan. Cada agente es responsable de un determinado aspecto de la funcionalidad y consta de tres componentes: Presentación, Abstracción y Control, que separan la interacción con el usuario de la funcionalidad central y la comunicación con otros agentes.
- ***Reflection:*** Proporciona un mecanismo para cambiar la estructura y el comportamiento del sistema dinámicamente. La aplicación se divide en dos partes: un meta-nivel y un nivel-base. El meta-nivel hace al software auto consciente.
- ***Microkernel:*** Separa un mínimo núcleo funcional de funcionalidad extendida y partes específicas del cliente. El *Microkernel* también sirve como punto donde engarzar estas piezas y coordinar su colaboración.

1.3.2.2 Patrones de diseño

Un patrón de este tipo identifica, abstrae y nombra los aspectos elementales de una estructura de diseño, donde los componentes, son las clases y objetos, y sus mecanismos de interacción son mensajes.

Cada patrón prescribe una estructura de clases, sus roles y colaboraciones, y una adecuada asignación de métodos para resolver un problema de diseño en una manera flexible y adaptable.

La aplicación de los patrones en un diseño consiste en identificar el patrón que resuelve el problema de diseño encontrado y aplicar la solución abstracta prescrita por el patrón a dicho problema.

Uno de los beneficios de utilizar patrones es el entendimiento y documentación de diseños orientados a objetos. Los patrones, también, mejoran el mantenimiento de sistemas ya que proveen una especificación explícita de clases e interacción entre objetos. Además, los patrones proveen un vocabulario para discutir y comunicar decisiones de diseño en término de estructuras de clases en lugar de objetos.

Los patrones de diseño ayudan a elegir diseños alternativos que hacen un sistema reutilizable y evitan alternativas que comprometan la reutilización.

Existen patrones que describen los principios fundamentales de diseño de objetos para la asignación de responsabilidades. Estos son los conocidos patrones GRASP, acrónimo de *General Responsability Assignment Software Patterns* (Patrones de Software para la asignación General de Responsabilidad).

Se pueden destacar 5 patrones principales que son:[13]

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

A continuación se exponen las características de cada uno de ellos:[13]

Experto

Problema: ¿Cómo asignar responsabilidades, de la forma más eficiente?

Solución: Asignar una responsabilidad al experto en información: la clase que cuenta con la información necesaria para cumplir la responsabilidad.

Si estas asignaciones de responsabilidades se hacen en la forma adecuada, los sistemas tienden a ser más fáciles de entender, mantener y ampliar, lo que nos ofrece la garantía de poder reutilizar los componentes en futuras aplicaciones.

Explicación: Experto es un patrón que se usa más que cualquier otro al asignar responsabilidades; es un principio básico que suele utilizarse en el diseño orientado a objetos. Con él no se pretende designar una idea oscura ni extraña; expresa simplemente la “intuición” de que los objetos hacen cosas relacionadas con la información que poseen.

Cuando el cumplimiento de una responsabilidad requiere de información distribuida en varias clases, aparecen “expertos parciales”, que colaboran con el cumplimiento de la responsabilidad en cuestión. En este caso de la existencia de “expertos parciales”, la interacción de estos se efectuará a través de mensajes para compartir el trabajo.

Este patrón como tantas otras cosas en la tecnología orientada a objetos, ofrece una analogía con el mundo real. Un ejemplo real sería el de una empresa, donde el director financiero para elaborar estado

financiero general necesita de la información de cuentas por cobrar y cuentas por pagar que le brindan los encargados de generar dichos reportes individuales sobre créditos y deudas.

Beneficios:

- Se conserva el encapsulamiento, ya que los objetos se valen de su propia información para hacer lo que se les pide. Esto soporta un bajo acoplamiento, lo que favorece el hecho de tener sistemas más robustos y de fácil mantenimiento (bajo acoplamiento es un patrón GRASP que se examinará más adelante).
- El comportamiento se distribuye entre las clases que cuentan con la información requerida, alentando con ello definiciones de clases “sencillas” y más cohesivas que son fáciles de comprender y mantener. Así se brinda una alta cohesión (patrón que se explicará más adelante).

Creador

Problema: ¿Quién debería ser el responsable de crear una nueva instancia de alguna clase?

Solución: La responsabilidades de crear una instancia de la clase A se le dará a aquella clase B, en los siguientes casos:

- B agrega los objetos A.
- B contiene los objetos A.
- B registra las instancias de los objetos A.
- B utiliza específicamente los objetos A.
- B tiene los datos de inicialización que serán transmitidos hacia A cuando este objeto sea creado (B es experto en la creación de A).

Explicación: El patrón Creador guía la asignación de responsabilidades relacionadas con la creación de objetos, tarea muy frecuente en los sistemas orientados a objetos. El propósito fundamental de este patrón es encontrar un creador que debemos conectar con el objeto producido en cualquier evento. Al escogerlo como creador, se da soporte al bajo acoplamiento.

El agregado agrega la parte, el contenedor contiene el contenido, el registro registra. En un diagrama de clases se registran las relaciones muy frecuentes entre las clases. El patrón Creador indica que

clase incluyente del contenedor o registro es idónea para asumir la responsabilidad de crear la cosa contenida o registrada. Desde luego, se trata tan sólo de una directriz.

Beneficios:

- Se brinda un soporte al bajo acoplamiento, lo cual supone menos dependencias respecto al mantenimiento y mejores oportunidades de reutilización. Es probable que el acoplamiento no aumente, pues la clase creada tiende a ser visible a la clase creador, debido a las asociaciones actuales que llevaron a elegirla como el parámetro adecuado.

Bajo Acoplamiento

Problema: ¿Cómo dar soporte a una dependencia escasa y a un aumento de la reutilización?

Solución: Asignar una responsabilidad para mantener bajo acoplamiento.

Explicación: El Bajo Acoplamiento es un principio que no podemos descartar durante las decisiones del diseño, como meta principal a lograr siempre. Se puede catalogar como un patrón evaluativo que el diseñador aplica al juzgar sus decisiones de diseño.

El bajo acoplamiento estimula la asignación de responsabilidades de forma tal que la inclusión de éstas no incremente el acoplamiento, creando clases más independientes y con mayor resistencia al impacto de los cambios, que aumentan la productividad y la posibilidad de reutilización.

Es válido aclarar que este patrón no puede verse de forma independiente a los patrones Experto y Alta Cohesión, sino más bien incluirse como otro de los principios del diseño que influyen de forma determinante a la hora de la asignación de responsabilidades.

Beneficios:

- No se afectan por cambios en otros componentes.
- Fáciles de entender por separado.
- Fáciles de reutilizar.

Alta Cohesión

Problema: ¿Cómo mantener la complejidad dentro de los límites manejables?

Solución: Asignar una responsabilidad de modo que la cohesión siga siendo alta (la cohesión es una medida de cuan relacionadas y enfocadas están las responsabilidades de una clase, además de que

una alta cohesión garantiza que clases con responsabilidades estrechamente relacionadas no realicen un trabajo enorme).

Explicación: Como el patrón Bajo Acoplamiento, también Alta Cohesión es un principio que debemos tener presente en todas las decisiones de diseño: es la meta principal que ha de buscarse en todo momento. Es un patrón evaluativo que el desarrollador aplica al valorar sus decisiones de diseño.

Ahora, existen formas de calificar el grado de cohesión, siendo estas: muy baja, baja, alta y moderada.

Un diseño con muy baja cohesión, es aquel en el que una clase es la responsable de operaciones correspondientes a áreas funcionales muy heterogéneas, sucediendo algo parecido con un diseño de baja cohesión, ya que en este caso la clase tiene la responsabilidad exclusiva de una tarea compleja dentro de un área funcional. En ninguno de los dos casos anteriores se delega o distribuyen las responsabilidades, dichas clases abarcan el volumen de las responsabilidades a realizar sin importar su complejidad o heterogeneidad.

Un diseño con un nivel de cohesión bajo, presenta problemas a la hora de comprender, reutilizar o conservar dichas clases, además de constantes afectaciones dadas por cambios en dicho diseño.

Un nivel moderado de cohesión implica que la clase posee un peso ligero pues agrupa áreas que están relacionadas lógicamente con el concepto de clases pero no entre ellas.

El alto nivel de cohesión, es presentado por las clases que tienen responsabilidades moderadas en un área funcional y colaboran con otras para llevar a cabo las tareas.

Una clase con mucha cohesión presenta beneficios tales como: facilidad de mantenimiento, comprensión y uso. Su alto grado de funcionalidad, combinada con una reducida cantidad de operaciones, también simplifica el mantenimiento y los mejoramientos. La ventaja que significa una gran funcionalidad también soporta un aumento de la capacidad de reutilización.

Beneficios:

- Mejoran la claridad y la facilidad con que se entiende el diseño.
- Se simplifican el mantenimiento y las mejoras en funcionalidad.
- A menudo se genera un bajo acoplamiento.
- La ventaja de una gran funcionalidad soporta una mayor capacidad de reutilización, porque una clase muy cohesiva puede destinarse a un propósito muy específico.

Controlador

Problema*:* ¿Quién debería encargarse de atender un evento del sistema?

Solución: Asignar la responsabilidad del manejo de un mensaje de los eventos del sistema a una clase que represente una de las siguientes opciones: Controlador de fachada, controlador de tareas o controlador de casos de uso.

Explicación: La mayor parte de los sistemas reciben eventos de entrada externa, los cuales generalmente incluyen una interfaz gráfica para el usuario operado por una persona. Otros medios de entrada son los mensajes externos entre ellos un conmutador de telecomunicaciones para procesar llamadas o las señales procedentes de sensores como sucede en los sistemas de control de procesos.

En todos los casos, si se recurre a un diseño orientado a objetos, hay que elegir los controladores que manejen esos eventos de entrada. Este patrón ofrece una guía para tomar decisiones apropiadas que generalmente se aceptan.

La misma clase controlador debería utilizarse con todos los eventos sistémicos de un caso de uso, de modo que podarnos conservar la información referente al estado del caso**.** Esta información será útil por ejemplo para identificar los eventos del sistema fuera de secuencia. Puede emplearse varios controladores en los casos de uso.

Beneficios:

- Mayor potencial de los componentes reutilizables. Garantiza que la empresa o los procesos de dominio sean manejados por la capa de los objetos de dominio y no por la de la interfaz.
- Reflexionar sobre el estado del caso de uso.

Un libro que ha popularizado los patrones de diseño es el conocido “*Design Patterns*”, escrito por los que comúnmente se conoce como GoF (*Gang of Four*, "Pandilla de los Cuatro"). En este se recopilan los patrones agrupados en tres categorías: de creación, de estructura y de comportamiento.

A continuación se listan estos patrones, con una breve descripción del propósito de cada uno de ellos:[4]

Patrones de creación

- ***Abstract Factory*:** Proporciona una interfaz para crear familias de objetos o que dependen entre sí, sin especificar sus clases concretas.

- ***Builder*:** Separa la construcción de un objeto complejo de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones.
- ***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.
- ***Prototype:*** Especifica los tipos de objetos a crear por medio de una instancia prototípica, y crear nuevos objetos copiando este prototipo.
- ***Singleton*:** Garantiza que una clase sólo tenga una instancia, y proporciona un punto de acceso global a ella.

Patrones estructurales

- ***Adapter:*** Convierte la interfaz de una clase en otra distinta que es la que esperan los clientes. Permite que cooperen clases que de otra manera no podrían por tener interfaces incompatibles.
- ***Bridge*:** Desvincula una abstracción de su implementación, de manera que ambas puedan variar de forma independiente.
- ***Composite:*** Combina objetos en estructuras de árbol para representar jerarquías de parte-todo. Permite que los clientes traten de manera uniforme a los objetos individuales y a los compuestos.
- ***Decorator*:** Añade dinámicamente nuevas responsabilidades a un objeto, proporcionando una alternativa flexible a la herencia para extender la funcionalidad.
- ***Facade:*** Proporciona una interfaz unificada para un conjunto de interfaces de un subsistema. Define una interfaz de alto nivel que hace que el subsistema se más fácil de usar.
- ***Proxy:*** Proporciona un sustituto o representante de otro objeto para controlar el acceso a éste.

Patrones de comportamiento

- ***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.

- ***Command:*** Encapsula una petición en un objeto, permitiendo así parametrizar a los clientes con distintas peticiones, encolar o llevar un registro de las peticiones y poder deshacer la operaciones.
- ***Interpreter:*** Dado un lenguaje, define una representación de su gramática junto con un intérprete que usa dicha representación para interpretar las sentencias del lenguaje.
- ***Iterator:*** Proporciona un modo de acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación interna.
- ***Mediator:*** Define un objeto que encapsula cómo interactúan un conjunto de objetos. Promueve un bajo acoplamiento al evitar que los objetos se refieran unos a otros explícitamente, y permite variar la interacción entre ellos de forma independiente.
- ***Memento:*** Representa y exporta el estado interno de un objeto sin violar la encapsulación, de forma que éste puede volver a dicho estado más tarde.
- ***Observer:*** Define una dependencia de uno a muchos entre objetos, de forma que cuando un objeto cambia de estado se notifica y actualizan automáticamente todos los objetos.
- ***State:*** Permite que un objeto modifique su comportamiento cada vez que cambia su estado interno. Parecerá que cambia la clase del objeto.
- ***Strategy:*** Define una familia de algoritmos, encapsula uno de ellos y los hace intercambiables. Permite que un algoritmo varíe independientemente de los clientes que lo usan.
- ***Template Method:*** Define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos. Permite que las subclases redefinan ciertos pasos del algoritmo sin cambiar su estructura.
- ***Visitor:*** Representa una operación sobre los elementos de una estructura de objetos. Permite definir una nueva operación sin cambiar las clases de los elementos sobre los que opera.

1.4 Lenguajes de Descripción Arquitectónica

Los lenguajes de descripción de arquitecturas ocupan una parte importante del trabajo arquitectónico desde la fundación de la disciplina. Se trata de un conjunto de propuestas de variado nivel de rigurosidad, casi todas ellas de extracción académica, que fueron surgiendo desde comienzos de la década de 1990 hasta la actualidad. Los ADLs permiten modelar una arquitectura mucho antes que se lleve a cabo la programación de las aplicaciones que la componen, analizar su adecuación, determinar

sus puntos críticos y eventualmente simular su comportamiento. Los ADLs son bien conocidos en los estudios universitarios de AS, pero muy pocos arquitectos de industria parecen conocerlos y son menos aún quienes los utilizan como instrumento en el diseño arquitectónico de sus proyectos. Hay unos veinte ADLs de primera magnitud y tal vez unos cuarenta o cincuenta propuestos en ponencias que no han resistido el paso del tiempo o que no han encontrado su camino en el mercado.[14]

Estos lenguajes permiten construir una estructura de alto nivel, es decir, sin detalles de implementación. Los ADLs son poco utilizados aunque algunos de ellos generan el *template* de la aplicación. La presencia de UML (lenguaje de modelado, será abordado en la siguiente sección) ha hecho que estos lenguajes no hayan ocupado el lugar que debían.

Entre las características a considerar de estos lenguajes se puede citar las siguientes:

- **Composición:** Permiten la representación del sistema como composición de una serie de partes.
- **Configuración:** La descripción de la arquitectura es independiente de la de los componentes que formen parte del sistema.
- **Abstracción:** Describen los roles o papeles abstractos que juegan los componentes dentro de la arquitectura.
- **Flexibilidad:** Permiten la definición de nuevas formas de interacción entre componentes.
- **Reutilización:** Permiten la reutilización tanto de los componentes como de la propia arquitectura.
- **Heterogeneidad:** Permiten combinar descripciones heterogéneas.
- **Análisis:** Permiten diversas formas de análisis de la arquitectura y de los sistemas desarrollados a partir de ella.

A continuación algunos de los más difundidos.

- **Acme-Armani:** Acme se define como una herramienta capaz de soportar el mapeo de especificaciones arquitectónicas entre diferentes ADLs, o en otras palabras, como un lenguaje de intercambio de arquitectura. No es entonces considerado por muchos un ADL en sentido estricto, sin embargo, posee numerosas prestaciones que también son propias de los ADLs. Se reconoce que como ADL no es necesariamente apto para cualquier clase de sistemas, al

mismo tiempo que se destaca su capacidad de describir con facilidad sistemas “relativamente simples”.

- **Jacal:** Es un lenguaje de descripción de arquitecturas de software de propósito general creado en la Universidad de Buenos Aires. El objetivo principal de Jacal es lo que actualmente se denomina “animación” de arquitecturas. Esto es, poder visualizar una simulación de cómo se comportaría en la práctica un sistema basado en la arquitectura que se ha representado. Más allá de este objetivo principal, el diseño de Jacal contempla otras características deseables en un ADL, como por ejemplo, contar con una representación gráfica que permita a simple vista transmitir la arquitectura del sistema sin necesidad de recurrir a información adicional. Mediante este procedimiento se puede comprobar o refutar propiedades deseables de los diseños y recopilar métricas dinámicas.[15]

- **CHAM:** Proporciona una base útil para la descripción de una arquitectura debido a su capacidad de componer especificaciones para las partes y describir explícitamente las reglas de composición. Es una técnica de especificación basada en álgebra de procesos que utiliza como fundamento teórico los sistemas de reescritura de términos para capturar la conducta comunicativa de los componentes arquitectónicos.[14]

Como resultado del estudio, y a pesar de las ventajas y utilidades que ofrecen los ADLs, no se propone la utilización de ellos por el esfuerzo que representaría capacitar personas en su uso, cuando existen otras formas de cumplir sus objetivos.

1.5 Lenguaje Unificado de Modelado (UML)

UML (*Unified Modeling Language*) es un lenguaje que permite modelar, construir y documentar los elementos que forman un sistema software orientado a objetos. Se ha convertido en el estándar de facto de la industria. Fue concebido Grady Booch, Ivar Jacobson y Jim Rumbaugh. Estos autores fueron contratados por la empresa *Rational Software Company* para crear una notación unificada en la que basar la construcción de sus herramientas CASE. En el proceso de creación de UML han participado, no obstante, otras empresas de gran peso en la industria como Microsoft, Oracle o IBM, así como grupos de analistas y desarrolladores.[16]

Los principales objetivos en el diseño de UML fueron: obtener un lenguaje simple pero suficientemente expresivo, que permitiese modelar aplicaciones en cualquier dominio; obtener un lenguaje legible, puesto que sería un lenguaje utilizado por las personas; y permitir la generación automática de código.

Ventajas de UML

UML ha mejorado el desarrollo de software no sólo al establecer un estándar común que simplifica la comunicación entre desarrolladores de software. Sus principios fundamentales son fáciles de entender y de aprender. Hoy en día, es el lenguaje de la ingeniería de software. Es utilizado no sólo para la especificación de un sistema sino también para propósitos de comunicación entre la gente involucrada en el desarrollo de un sistema (ingenieros, científicos del área de computación, administradores, líderes y otros), o para la documentación de software existente.[16]

Como resultado de las ventajas que ofrece UML, se seleccionó como lenguaje de modelado. Además, por poseer un propósito general y comprensible, se propone para describir la arquitectura, a pesar de no ser propiamente un ADL.

1.6 Metodología de desarrollo de software

Las metodologías de desarrollo de software son un conjunto de procedimientos, técnicas y ayudas a la documentación para el desarrollo de productos software.[17]

Una metodología de desarrollo de software indica paso a paso todas las actividades a realizar para lograr el producto informático deseado, indicando además qué personas deben participar en el desarrollo de las actividades y qué papel deben tener. Además detallan la información que se debe producir como resultado de una actividad y la información necesaria para comenzarla.

Rational Unified Process (RUP)

El Proceso Unificado de Racional es una metodología guiada por casos de uso, centrado en la arquitectura, iterativo e incremental. Su desarrollo está basado en componentes. RUP contiene un proceso integrado y propone un modelo de referencia organizacional del personal. Utiliza UML como fundamental lenguaje de modelado para el desarrollo de todos los modelos.[18]

RUP divide el proceso de desarrollo en ciclos, teniendo un producto al finalizar cada uno, cada ciclo se divide en las siguientes fases:[18]

- **Inicio:** El objetivo en esta etapa es determinar la visión del proyecto.
- **Elaboración:** En esta etapa el objetivo es determinar la arquitectura óptima.
- **Construcción:** En esta etapa el objetivo es llevar a obtener la capacidad operacional inicial.
- **Transición:** El objetivo es llegar a obtener el *release* del proyecto.

Cada una de estas etapas es desarrollada mediante el ciclo de iteraciones, el cual consiste en reproducir el ciclo de vida en cascada a menor escala. Los objetivos de una iteración se establecen en función de la evaluación de las iteraciones precedentes.

Vale mencionar que el ciclo de vida que se desarrolla por cada iteración, es llevado bajo dos disciplinas:

- Disciplina de Desarrollo
    - Ingeniería de Negocios: Entendiendo las necesidades del negocio.
    - Requerimientos: Trasladando las necesidades del negocio a un sistema automatizado.
    - Análisis y Diseño: Trasladando los requerimientos dentro de la arquitectura de software.
    - Implementación: Creando software que se ajuste a la arquitectura y que tenga el comportamiento deseado.
    - Pruebas: Asegurándose que el comportamiento requerido es el correcto y que todo lo solicitado está presente.
- Disciplina de Soporte
    - Configuración y administración del cambio: Guardando todas las versiones del proyecto.
    - Administrando el proyecto: Administrando horarios y recursos.
    - Ambiente: Administrando el ambiente de desarrollo.
    - Distribución: Hacer todo lo necesario para la salida del proyecto.

Proceso unificado abierto (OpenUp/Basic)

OpenUP es un proceso de desarrollo de software de código abierto que aplica propuestas iterativas e incrementales dentro del ciclo de vida, tratando de ser manejable en relación con el RUP. Se valora la colaboración y el aporte de los *stakeholders* sobre los entregables y la formalidad innecesarios.

OpenUP está organizado dentro de cuatro áreas principales de contenido: Comunicación y Colaboración, Intención, Solución, y Administración.

OpenUP se caracteriza por cuatro principios básicos que se soportan mutuamente:

- Colaboración para alinear los intereses y un entendimiento compartido: el software es creado por personas con diferentes intereses y habilidades quienes trabajan juntos para crear software eficientemente.

  Hay que desarrollar prácticas que fomenten un ambiente de equipo saludable que permita la colaboración efectiva, lo cual encuadra los intereses de los participantes del proyecto (equipo de desarrollo, aseguramiento de la calidad, *stakeholders* del producto, clientes) y ayude a los participantes del proyecto a desarrollar un entendimiento compartido del proyecto.

- Balance para confrontar las prioridades (necesidades y costos técnicos) para maximizar el valor para los *stakeholders*: los participantes del proyecto y los *stakeholders* deben colaborar para desarrollar una solución que maximice el beneficio y cumpla con las restricciones planteadas en el proyecto. Lograr un balance es un proceso dinámico porque si los *stakeholders* y los participantes del proyecto aprenden más acerca del sistema, entonces sus prioridades y restricciones cambian.

- Enfoque en articular la arquitectura para facilitar la colaboración técnica, reducir los riesgos y minimizar excesos y trabajo extra: sin un fundamento arquitectónico, un sistema evolucionará en una forma ineficiente y casual. Tal sistema frecuentemente presenta dificultades para evolucionar, reutilizarse o integrarse sin una reconstrucción sustancial. Esto también dificulta organizar el equipo o comunicar las ideas sin el enfoque técnico común que la arquitectura proporciona.

  Hay que usar la arquitectura como un punto focal para que los desarrolladores alineen sus intereses e ideas, articulando las decisiones técnicas esenciales a través de una arquitectura creciente.

- Evolución continua para reducir riesgos, demostrar resultados y obtener retroalimentación de los clientes: usualmente no es posible conocer todas las necesidades de los *stakeholders*, ser consciente de todos los riesgos, comprender todas las tecnologías del proyecto, o saber como trabajar en equipo. Aún si fuese posible conocer todas estas cosas, es probable que cambien durante la vida del proyecto.

  Se dividirá el proyecto en proyectos más pequeños, iteraciones enmarcadas en tiempo para demostrar valor incremental y obtener retroalimentación temprana y continua.

La metodología seleccionada fue OpenUp por ser un proceso ágil y ligero que promueve el desarrollo del software a través de las mejores prácticas, haciéndolo un proceso pequeño y extensible (si es

necesario). Debido a sus características y a que se utiliza para proyectos pequeños, fue decisión del polo de Bioinformática utilizar esta metodología.

1.7 Herramienta CASE

CASE corresponde a las iniciales de: *Computer Aided Software Engineering*; y en su traducción al Español significa Ingeniería de Software Asistida por Computadora.[19]

El concepto de CASE es muy amplio; y una buena definición genérica, que puede abarcar esa amplitud de conceptos, sería la de considerar a la Ingeniería de Software Asistida por Computación, como la aplicación de métodos y técnicas a través de las cuales las personas pueden modelar o diseñar sistemas por medio de programas, procedimientos y su respectiva documentación.[19]

Rational Rose

Rational Rose es la herramienta CASE desarrollada por los creadores de UML (Booch, Rumbaugh y Jacobson), que cubre todo el ciclo de vida de un proyecto: concepción y elaboración del modelo, construcción de los componentes, transición a los usuarios. Permite especificar, analizar, diseñar el sistema antes de codificarlo.[20]

Rational Rose ofrece:

- Mantener la consistencia de los modelos del sistema software.
- Chequeo de la sintaxis UML.
- Generación de documentación automáticamente.
- Generación de código a partir de los modelos.
- Ingeniería inversa (crear modelo a partir código).
- Soporte de ingeniería *Forward* y/o reversa para algunos de los conceptos más comunes de Java.
- El Add-In para modelado Web provee visualización, modelado y las herramientas para desarrollar aplicaciones de Web.
- Modelado UML para trabajar en diseños de base de datos, con capacidad de representar la integración de los datos y los requerimientos de aplicación a través de diseños lógicos y físicos.

Visual Paradigm

Visual Paradigm para UML es una herramienta UML fácil de usar que soporta la última notación UML 2.1, ingeniería inversa, generación de código, importación desde Rational Rose, exportación/importación XML, generador de informes, editor de figuras, integración con Microsoft Visio, plug-in, integración IDE con Visual Studio, Eclipse, NetBeans y otros. Entre sus características se incluyen el modelado colaborativo con CVS y Subversion, interoperabilidad con modelos UML2 a través de XMLI.

Visual Paradigm ofrece:

- Diseño centrado en casos de uso y enfocado al negocio que generan un software de mayor calidad
- Uso de un lenguaje estándar común a todo el equipo de desarrollo que facilita la comunicación.
- Modelo y código que permanece sincronizado en todo el ciclo de desarrollo
- Disponibilidad de múltiples versiones, para cada necesidad.
- Disponibilidad de integrarse en los principales IDEs.
- Disponibilidad en múltiples plataformas

La herramienta CASE seleccionada fue Visual Paradigm. Además de su potencia y de utilizar como lenguaje de modelado UML, permite diseñar un producto con calidad y de forma rápida. Por sus características y por contar la Universidad con la licencia para su uso, se considera la más adecuada para la modelación de las vistas de la arquitectura y para los demás artefactos generados en el proyecto.

1.8 Lenguajes, tecnologías y herramientas de soporte al desarrollo

La selección del lenguaje, tecnologías y herramientas a utilizar para el desarrollo de una aplicación es sumamente importante y determina el tiempo de desarrollo, los costos y el resultado general de todo el proceso.

1.8.1 Lenguaje de programación

En la actualidad se ha extendido el uso de los lenguajes de programación orientados a objetos, por sus ventajas y beneficios. La programación orientada a objetos promete mejoras de amplio alcance en la forma de diseño, desarrollo y mantenimiento del software. Ofrece una solución a largo plazo a los

problemas y preocupaciones que han existido desde el comienzo en el desarrollo de software: la falta de portabilidad del código y reusabilidad, código que es difícil de modificar, ciclos de desarrollo largos y técnicas de codificación no intuitivas. Permite crear sistemas más complejos, relacionar el sistema al mundo real y facilita la creación de programas visuales.

Lo interesante de la POO es que proporciona conceptos y herramientas con las cuales se modela y representa el mundo real tan fielmente como sea posible.

C#

C# (leído en inglés "C Sharp" y en español "C Almohadilla") es un lenguaje de propósito general diseñado por Microsoft para la plataforma .NET. Sus principales creadores son Scott Wiltamuth y Anders Hejlsberg, éste último también conocido por haber sido el diseñador del lenguaje Turbo Pascal y la herramienta RAD Delphi. A continuación se enuncian sus principales características:[21]

- **Orientado a Objetos:** Soporta todas las características propias del paradigma de programación orientada a objetos: encapsulación, herencia y polimorfismo. Es más puro respecto a otros lenguajes como C++, ya que no admite ni funciones ni variables globales sino que todo el código y datos han de definirse dentro de definiciones de tipos de datos, lo que reduce problemas por conflictos de nombres y facilita la legibilidad del código.
- **Seguridad de tipos:** Incluye mecanismos que permiten asegurar que los accesos a tipos de datos siempre se realicen correctamente, lo que permite evitar que se produzcan errores difíciles de detectar por acceso a memoria no perteneciente a ningún objeto y es especialmente necesario en un entorno gestionado por un recolector de basura.
- **Modernidad:** Incorpora en el propio lenguaje elementos que a lo largo de los años ha ido demostrándose son muy útiles para el desarrollo de aplicaciones y que en otros lenguajes como C++ hay que simular. Incluye una instrucción *foreach* que permita recorrer colecciones con facilidad y es ampliable a tipos definidos por el usuario y un tipo básico *string* para representar cadenas o la distinción de un tipo *bool* específico para representar valores lógicos.
- **Extensibilidad de tipos básicos:** Permite definir, a través de estructuras, tipos de datos para los que se apliquen las mismas optimizaciones que para los tipos de datos básicos, es decir, que se puedan almacenar directamente en pila y se asignen por valor y no por referencia.

Java

Ofrece toda la funcionalidad de un lenguaje potente, pero sin las características menos usadas y más confusas. Es orientado a objetos, trabaja con sus datos como objetos y con interfaces a esos objetos. Está diseñado para ser lo suficientemente simple para que los programadores puedan lograr fluidez con el lenguaje. Proporciona las librerías y herramientas para que los programas puedan ser distribuidos, es decir, que se corran en varias máquinas, interactuando.

- **Orientado a Objetos:** Soporta las características esenciales del paradigma de la programación a objetos: encapsulación, herencia y polimorfismo. Java hace uso de la definición de entidades formadas por métodos y variables que reciben el nombre de clases, la instancia de alguna clase cualquiera en Java recibe el nombre de objeto. [22]

- **Robusto:** Elimina el uso de apuntadores para referenciar localidades de memoria, además libera al desarrollador de la necesidad de desalojar la memoria que la aplicación ya no usa, aunado a esto, requiere la declaración explícita tanto de los tipos de datos como de métodos. Estos factores que se acaban de mencionar, destacan como causas comunes de errores lo que resulta en aplicaciones poco fiables. Al implementar una aplicación en Java se reduce el porcentaje de errores ocasionados por las causas antes mencionadas lo que da como consecuencia programas más robustos y confiables. Finalmente Java hace uso de excepciones para notificar al usuario acerca de las fallas que puedan encontrarse en el sistema y recuperar dicho sistema. [22]

- **Multiplataforma:** El mismo código Java que funciona en un sistema operativo, funcionará en cualquier otro sistema operativo que tenga instalada la máquina virtual de Java.[23]

- **Seguro:** La máquina virtual, al ejecutar el código Java, realiza comprobaciones de seguridad, además el propio lenguaje carece de características inseguras, como por ejemplo los punteros.[23]

- **Multitarea:** A pesar de que las capacidades de multitarea que pueden ser implementadas en Java dependen en gran parte del sistema operativo en el cual se ejecuten, dígase Windows o Unix, dichas capacidades superan en gran manera a los entornos de flujo único (single-thread) que ofrecen otros lenguajes de programación. Al ser multitarea, Java permite la ejecución concurrente de varios procesos ligeros o hilos de ejecución.[22]

- **Simple:** El único requerimiento para aprender Java es tener una comprensión de los conceptos básicos de la programación orientada a objetos. Así se ha creado un lenguaje simple (aunque eficaz y expresivo) pudiendo mostrarse cualquier planteamiento por parte del programador.

Para el desarrollo de la plataforma se seleccionó el lenguaje de programación orientado a objeto Java por ser multiplataforma, característica imprescindible que se requiere para la aplicación. Por otro lado influyó en la selección, la existencia de una primera versión programada en Java, permitiendo de esta forma la reutilización de gran cantidad de código.

1.8.2 Sistema de Control de Versiones

Un sistema de control de versiones es un sistema de gestión de archivos y directorios, cuya principal característica es que mantiene la historia de los cambios y modificaciones que se han realizado sobre ellos a lo largo del tiempo. De esta forma, el sistema es capaz de “recordar” las versiones antiguas de los datos, lo que nos permite examinar el histórico de cambios o recuperar versiones anteriores de un fichero, incluso aunque haya sido borrado.[24]

Concurrentes (CVS)

Es un sistema cliente-servidor que permite a los desarrolladores realizar el seguimiento de las diferentes versiones del código fuente de un proyecto. Su uso está muy extendido entre la comunidad de desarrolladores, empleándose desde hace años en los proyectos de código abierto del W3C, SourceForge, Apache o GNU entre otros. Es una herramienta que facilita:[24]

- Registrar todos los cambios efectuados sobre los archivos de un proyecto.
- Recuperar versiones anteriores del código de un proyecto.
- Conocer qué cambios se han efectuado sobre un archivo determinado, quién los ha realizado y cuándo.
- Gestionar los conflictos que pueden producirse en entornos en los que los desarrolladores se encuentran distribuidos geográficamente.

Subversion (SVN)

Subversion es un sistema de control de versiones que se ha popularizado bastante, en especial dentro de la comunidad de desarrolladores de software libre. Está preparado para funcionar en red, y se distribuye bajo una licencia libre de tipo Apache.

El sistema de control de versiones Subversion presenta mejoras frente al sistema de control de versiones concurrentes como son:[24]

- Mantiene versiones no sólo de archivos, sino también de directorios.
- Mantiene versiones de los meta datos asociados a los directorios.
- Además de los cambios en el contenido de los documentos, se mantiene la historia de todas las operaciones de cada elemento, incluyendo la copia, cambio de directorio o de nombre.
- Atomicidad de las actualizaciones. Una lista de cambios constituye una única transacción o actualización del repositorio. Esta característica minimiza el riesgo de que aparezcan inconsistencias entre distintas partes del repositorio.
- Posibilidad de elegir el protocolo de red. Además de un protocolo propio (svn), puede trabajar sobre http (o https). La capacidad de funcionar con un protocolo tan universal como el http simplifica la implantación (cualquier infraestructura de red actual soporta dicho protocolo) y universaliza las posibilidades de acceso (si se quiere, puede utilizarse a través de Internet).
- Soporte tanto de ficheros de texto como de binarios.
- Mejor uso del ancho de banda, ya que en las transacciones se transmiten sólo las diferencias y no los archivos completos.
- Mayor eficiencia en la creación de ramas y etiquetas que en CVS.

Después de analizar los elementos planteados anteriormente, se decidió usar el sistema de control de versiones Subversion en el entorno en el que se desarrolla la plataforma. Es importante aprovechar los beneficios de esta herramienta al reducir las inconsistencias y agilizar las transferencias y actualizaciones de datos porque al estar dividido el trabajo en varios módulos, es necesario que los equipos tengan disponible y actualizada las informaciones que requieren. Subversion, al ahorrar recursos en la transferencia de datos, contribuye a que haya más recursos disponibles para las demás herramientas, que son numerosas y requieren capacidades.

1.8.3 Entorno integrado de desarrollo

Un entorno de desarrollo integrado o en inglés *Integrated Development Environment* (IDE) es un programa compuesto por un conjunto de herramientas para un programador. Puede dedicarse en exclusiva a un sólo lenguaje de programación o bien, poder utilizarse para varios. Un IDE es un

entorno de programación que ha sido empaquetado como un programa de aplicación, es decir, consiste en un editor de código, un compilador, un depurador y un constructor de interfaz gráfica GUI.

NetBeans

Este IDE es una herramienta para programadores, para escribir, compilar, depurar y ejecutar programas. Está escrito en Java pero puede servir para cualquier otro lenguaje de programación. Existe además un número importante de módulos para extender el IDE NetBeans, es un producto libre y gratuito sin restricciones de uso.

JBuilder

Es un entorno de desarrollo integrado para el lenguaje de programación Java desarrollado por Borland. Posee varias ediciones, la Enterprise, para aplicaciones J2EE, Web Services y struts, la Developer, para el desarrollo completo de aplicaciones Java, y la Foundation, con capacidades básicas para iniciarse en Java. Este IDE de desarrollo solo puede ser ejecutado sobre el sistema operativo Windows.

Eclipse

En la web oficial de Eclipse se define como "*An IDE for everything and for nothing in particular*" (un IDE para todo y para nada en particular). El IDE Eclipse es, únicamente, una de las herramientas que se engloban bajo el denominado Proyecto Eclipse.[25]

Es una plataforma de desarrollo extensible, basada en Java y de tipo *open-source*. Este entorno de desarrollo integrado ofrece, el control del editor de código, del compilador y del depurador desde una única interfaz de usuario. Su misión consiste en evitar tareas repetitivas, facilitar la escritura de código correcto, disminuir el tiempo de depuración e incrementar la productividad del desarrollador.

Para el desarrollo de la aplicación es necesario escoger el IDE a utilizar según el lenguaje de programación seleccionado, por lo tanto se hará uso del Eclipse. Este IDE es compatible con Java, permite la integración del entorno de desarrollo con Subversion y es multiplataforma, características que definen su selección.

1.8.4 Plataforma

En la informática, una plataforma de desarrollo es el entorno de software común en el cual se desenvuelve la programación de un grupo definido de aplicaciones. Comúnmente se encuentra

relacionada directamente a un sistema operativo; sin embargo, también es posible encontrarla ligada a una familia de lenguajes de programación.

Dentro de las plataformas disponibles para el lenguaje de programación seleccionado, se encuentran:

JSE

La JSE (*Java Standard Edition*) permite desarrollar y desplegar aplicaciones de Java tanto en computadoras de escritorio como servidores, así como ambientes que demandan ejecución en tiempo real. La plataforma JSE proporciona las bases para la creación de la plataforma JEE (*Java Enterprise Edition*).

Hay dos productos principales en la familia de la plataforma JSE de Java: el JRE (*Java Runtime Environment*) y el JDK (*Java Development Kit*). El JSE provee las librerías, la máquina virtual y otros componentes que permiten la ejecución de los programas escritos en Java. El JDK provee los compiladores y otras herramientas necesarias para desarrollar los programas en Java.

Esta plataforma es también conocida como la Plataforma J2SDK.

JEE

La plataforma JEE ha sido diseñada para aplicaciones distribuidas con base en componentes o unidades funcionales de software que interactúan entre sí para formar parte de una aplicación empresarial. Un componente de esta plataforma debe formar parte de una aplicación y ser desplegado en un contenedor, o sea, en la parte del servidor JEE que le ofrece al componente ciertos servicios de bajo nivel y de sistema, tales como seguridad, manejo de concurrencia, persistencia y transacciones. Como se puede apreciar, JEE no es solo una plataforma o una tecnología, sino un estándar de desarrollo, construcción y despliegue de aplicaciones. Esta plataforma constituye un estándar para la implementación de servicios Web y arquitecturas SOA.

Esta plataforma es también conocida como la Plataforma J2EE.

JME

La plataforma JME (*Java Micro Edition*) proporciona un ambiente robusto y para la ejecución de aplicaciones que funcionan en dispositivos móviles, como son los teléfonos celulares, PDAs (*Personal Digital Aassistants*). JME incluye interfaces flexibles, seguridad robusta, protocolos de red incorporados, y la ayuda para las aplicaciones conectados y fuera de línea que se pueden transferir dinámicamente. Las aplicaciones basadas en JME son portables a través de muchos dispositivos.

Se propone el uso de la plataforma JEE porque crea un estándar en el cual los componentes de la aplicación pueden ser distribuidos y reutilizados. El uso de esta plataforma influye en la creación de aplicaciones portables, permite construir aplicaciones que se puedan ejecutar en cualquier tipo de plataforma o sistema operativo. Es una plataforma de desarrollo madura bien documentada. Además, incluye las funcionalidades de la JSE.

1.8.5 Frameworks

Los *frameworks* son soluciones que implementan patrones y ayudan a desarrollar funciones dentro de las aplicaciones a un nivel superior. Tienen la funcionalidad de abstraer de complejas implementaciones y hacer el desarrollo más ágil, ya que proveen soluciones a problemas comunes. Constituyen elementos a tener en cuenta a la hora de modelar una arquitectura pues hacen que se cubran en gran medida los objetivos con los que debe cumplir la misma**.** Entre los *frameworks* más populares para el lenguaje Java se encuentran los siguientes:

Struts

Recomendado para aplicaciones de gran tamaño y complejidad, posee un conjunto de clases que usan los estándares JEE (Servlets, JSP, Tags Personalizados, XML) y ayudan a hacer el desarrollo más rápido cuando se trata de una aplicación Web. Struts está basado en la arquitectura MVC, lo que garantiza la separación de cada una de estas capas, con vistas a lograr una aplicación escalable, reutilizable y con un nivel alto de profesionalidad. [26]

Java Server Faces (JSF, o simplemente “Faces”)

Posibilita desarrollar aplicaciones Web de una manera fácil y brinda poderosos componentes de interfaz de usuario, tal es el caso de textboxes, listboxes, paneles y datagrids. Brinda una arquitectura de componentes, un grupo de elementos para la interfaz de usuario y una infraestructura de aplicación. Posee una poderosa paleta de componentes orientados a eventos, lo cual permite un manejo de estos.[27]

Spring

Centrado a manejar los objetos de negocio. Facilita el desarrollo de buenas prácticas de programación. Ayuda a resolver muchos problemas evitando el uso de los EJB, provee una alternativa a los mismos que es apropiada para muchas aplicaciones. Spring puede usar Programación Orientada a Aspectos (AOP, por sus siglas en inglés) para el manejo de transacciones, seguridad, sin tener que usar el contendor de EJB.[28]

Hibernate

Hace posible el manejo de la capa de persistencia de cualquier aplicación. Realiza el mapeo entre el mundo orientado a objetos de las aplicaciones y el mundo entidad-relación de las bases de datos en entornos Java. El término utilizado es ORM (*object/relational mapping*) y consiste en la técnica de realizar la transición de una representación de los datos de un modelo relacional a un modelo orientado a objetos y viceversa. Constituye un motor de persistencia que implementa múltiples funcionalidades.[29]

Luego de realizar el estudio se decide para el desarrollo de la plataforma el uso del *framework* Hibernate. Dentro de las características que determinaron su selección se encuentran:

Presenta un mecanismo persistente totalmente transparente, los objetos desconocen su capacidad de persistir, no necesitan extender comportamientos especiales de otra clase o interfaz, no necesitan un contenedor de aplicaciones, pueden tener lógica de aplicación, son de fácil manejo y pueden ser usados para transportar los datos a cualquier capa de la aplicación.

Este *framework* es una clara implementación del patrón DAO. Este patrón permite contar con diversas fuentes de datos de tal forma que se encapsula la forma de acceder a estos. Hibernate crea una capa separada que se ocupa del acceso a datos con total independencia del gestor y la base de datos, dando la oportunidad de trabajar con varios gestores y bases de datos dentro de la misma aplicación sin que esto cree ningún conflicto en el modelo de objetos.

Actualmente en las aplicaciones informáticas, el acceso a los datos representa un serio problema, el *framework* Hibernate resultó el seleccionado pues proporciona una solución factible a dicho inconveniente. Este framework crea un puente entre el mundo orientado a objetos de las aplicaciones y el entorno relacional de las bases de datos, abstrayendo a los desarrolladores de las consultas SQL. Hibernate no necesita ningún entorno especial para ejecutarse.

1.8.6 Gestor de Base de datos

Un Sistema de Gestión de Bases de Datos (SGBD) consiste en una colección de datos interrelacionados y un conjunto de programas para acceder a esos datos. El objetivo primordial de un SGBD es proporcionar un entorno que sea a la vez conveniente y eficiente para ser utilizado al extraer y almacenar información de la base de datos.[30]

MySQL

MySQL es un sistema de gestión de bases de datos relacional, licenciado bajo la GPL de la GNU. Su diseño multihilo le permite soportar una gran carga de forma muy eficiente.

Este gestor de bases de datos es muy usado en el mundo del software libre, debido a su gran rapidez y facilidad de uso. Esta gran aceptación se basa, en parte, a que existen infinidad de librerías y otras herramientas que permiten su uso a través de gran cantidad de lenguajes de programación, además de su fácil instalación y configuración.

MySQL presenta un gran número de características como[31]:

- Aprovecha la potencia de sistemas multiprocesadores, gracias a su implementación multihilo.
- Soporta gran cantidad de tipos de datos para las columnas.
- Dispone de API's en gran cantidad de lenguajes (C, C++, Java, PHP).
- Gran portabilidad entre sistemas.
- Soporta hasta 32 índices por tabla.
- Gestión de usuarios y contraseñas, manteniendo un muy buen nivel de seguridad en los datos.

PostgreSQL

PostgreSQL es un producto vanguardia en el mercado, que a diferencia de MySQL soporta tanto bases de datos relacionales como orientadas a objetos. Otra particularidad de PostgreSQL es ser *Open Source* bajo licencia BSD por lo que sus potencialidades están en constante perfeccionamiento, característica que lo sitúa por delante de otros productos competitivos como Oracle o Microsoft SQL Server 2000, ambos; software propietarios. Funciona en todos los sistemas operativos importantes, incluyendo Linux, UNIX y Windows. Tiene soporte total para *foreign keys, joins, views, triggers*, y *stored procedures* (en múltiples lenguajes). Incluye la mayoría de los tipos de datos como son *INTEGER, NUMERIC, BOOLEAN, CHAR, VARCHAR, DATE* e *INTERVAL*.[32]

PostgreSQL está ampliamente considerado como el sistema de bases de datos de código abierto más avanzado del mundo. Presenta un gran número de características como:[33]

- **DBMS Objeto-Relacional**. PostgreSQL aproxima los datos a un modelo objeto-relacional, y es capaz de manejar complejas rutinas y reglas. Ejemplos de su avanzada funcionalidad son

consultas SQL declarativas, control de concurrencia multi-versión, soporte multi-usuario, optimización de consultas, herencia y arreglos.

- **Altamente Extensible:** Soporta operadores, funciones, métodos de acceso y tipos de datos definidos por el usuario.
- **Soporte SQL Comprensivo:** Soporta la especificación SQL99 e incluye características avanzadas tales como las uniones *(joins)* SQL92.
- **Integridad Referencial:** Soporta integridad referencial, la cual es utilizada para garantizar la validez de los datos de la base de datos.
- **API Flexible:** La flexibilidad del API de PostgreSQL ha permitido a los vendedores proporcionar soporte al desarrollo fácilmente para el RDBMS PostgreSQL. Estas interfaces incluyen Object Pascal, Python, Perl, PHP, ODBC, Java/JDBC, C/C++.
- **Lenguajes Procedurales:** PostgreSQL tiene soporte para lenguajes procedurales internos, incluyendo un lenguaje nativo denominado PL/pgSQL. Este lenguaje es comparable al lenguaje procedural de Oracle, PL/SQL. Otra ventaja de PostgreSQL es su habilidad para usar Perl, Python, o TCL como lenguaje procedural embebido.

El gestor de base de datos seleccionado fue PostgreSQL ya que a pesar de consumir más recursos y ser más lento que MySQL, posee una gran escalabilidad (característica que no implementa MySQL). Es capaz de ajustarse al número de CPUs y a la cantidad de memoria que posee el sistema de forma óptima, haciéndolo capaz de soportar una mayor cantidad de peticiones simultáneas. MySQL ofrece mayor velocidad para bases de datos pequeñas y que no requieran integridad ni restricciones *(constrains)*, pero para cualquier otra base de datos como la de la plataforma bioGRATO que puede alcanzar gran tamaño, PostgreSQL ofrece mejores rendimientos, incluso mejor velocidad en consultas compuestas.

1.8.7 Servidor de Aplicaciones

Se denomina servidor de aplicaciones a algunos servidores Web de nueva generación que proporcionan la lógica de negocio sobre la que construir aplicaciones. Suelen asociarse con servidores de alto rendimiento pensados para dar servicio a sitios Web con grandes necesidades: afluencia de visitas, movimiento de datos, atención de transacciones hacia bases de datos. Generalmente los fabricantes del sector tienen a disposición del público un servidor Web básico y otro con multitud de extensiones fuertemente integradas al que llaman servidor de aplicaciones.[34]

Tomcat

Tomcat (también llamado *Jakarta Tomcat* o *Apache Tomcat*) funciona como un contenedor de *servlets* desarrollado bajo el *proyecto Jakarta* en la *Apache Software Foundation*. Tomcat implementa las especificaciones de los *servlets* y de *JavaServer Pages* (*JSP*) de *Sun Microsystems*. Es un servidor de aplicaciones que implementa las tecnologías Java para servlets y páginas JSP. Un servidor de aplicaciones, a diferencia de un servidor Web, como es Apache, incluye un contenedor Web que permite servir páginas dinámicas.

Fueron valorados varios servidores de aplicaciones, entre ellos JBoss, Sun Application Server y Websphere Application Server. El servidor Tomcat es más liviano en el sentido que ocupa menos memoria y se puede levantar y bajar más rápidamente. Dado que Tomcat fue escrito en Java, funciona en cualquier sistema operativo que disponga de la máquina virtual Java.

Tomcat no funciona con cualquier servidor web con soporte para *servlets* y *JSPs*. Tomcat incluye el compilador *Jasper*, que compila *JSPs* convirtiéndolas en *servlets*. El motor de *servlets* del Tomcat a menudo se presenta en combinación con el servidor web *Apach*e.

Se seleccionó el Tomcat como servidor de aplicaciones pues entre otras características, puede funcionar como servidor Web por sí mismo. Este consume menos recursos que otros servidores de aplicaciones; es gratis y cuenta con un gran número de usuarios y soporte en la comunidad mundial.

1.9 El arquitecto de software

El rol de Arquitecto de Software es el de liderar y coordinar actividades técnicas y artefactos durante el proyecto. Se encarga de la definición y documentación de la arquitectura que guiará el desarrollo, y de la continua refinación de la misma en cada iteración; debe construir cualquier prototipo necesario para probar aspectos riesgosos desde el punto de vista técnico del proyecto; definirá los lineamientos generales del diseño y la implementación. Es responsable de diseñar las vistas que definen los requisitos, el diseño, la implementación y el despliegue del sistema.

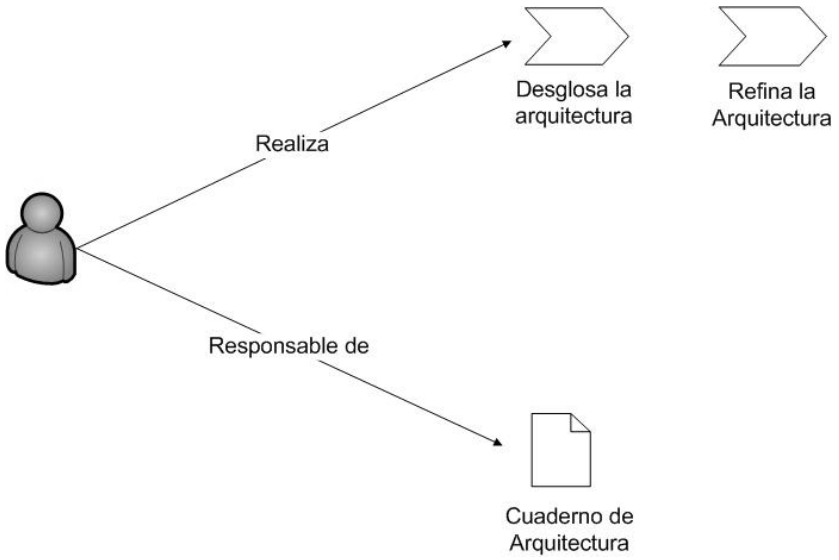

Figura 1. Funciones del arquitecto según OpenUP.

Como se muestra en la figura 1, el arquitecto es quien realiza las actividades de desglose de la arquitectura y refinamiento de la arquitectura; además, es el responsable del cuaderno de arquitectura.

El cuaderno de arquitectura describe el contexto de desarrollo del software. Contiene las decisiones, los fundamentos, las hipótesis, las explicaciones y las consecuencias de la formación de la arquitectura. Tiene como propósito lograr la integridad y comprensibilidad del sistema. Orienta a los desarrolladores que van a construir el sistema, constituye un artefacto crítico utilizado para tomar decisiones arquitectónicas, las cuales son explicadas a los desarrolladores. En general guía a los desarrolladores en la construcción del sistema, pero no contiene información de diseño aunque hace referencia a elementos de diseño arquitectónicamente significativos.

El arquitecto también debe participar en otras actividades, como son: el desarrollo del documento Visión, captura y descripción de requisitos, realización del plan de proyecto y plan de iteración, el diseño de la solución y la evaluación de los resultados.

1.10 Características del Sistema

El sistema que se desea construir, constituirá una potente herramienta para los especialistas químicos, pues agrupará un conjunto de operaciones que son necesarias para la investigación y elaboración de nuevos fármacos. El sistema debe permitir de manera integrada, la realización de diferentes operaciones, entre las que se encuentran: predicción de actividad biológica por medio de diversas técnicas de Inteligencia Artificial, entre ellas Lógica Difusa (LD) y Máquinas de Soporte Vectorial (SVM). Además debe permitir la fragmentación y el cálculo de descriptores, visualización y edición de

las moléculas, búsqueda de fragmentos de moléculas en dos y tres dimensiones, entre otras. Todas estas operaciones deben estar integradas en un único entorno, y de esta forma permitir un mejor desempeño del usuario, quien podrá realizar todas sus investigaciones en una misma computadora.

Para el correcto funcionamiento del sistema, debe permitir al especialista químico adicionar o quitar funcionalidades, razón por la cual debe funcionar y aportar resultados independientes. Además, deberá permitir la interacción entre cada parte del sistema que represente una funcionalidad, para permitir que las salidas de una operación puedan ser utilizadas como entradas para otras.

1.11 Conclusiones

En este capítulo se han abordado temas relacionados con el objeto de investigación, tendencias, metodologías y las tecnologías más utilizadas en el mundo informático, con el objetivo de proporcionar una solución que permita la gestión de información y comunicación entre los módulos de la plataforma. Después de haber realizado el estudio de cada uno de estos temas y teniendo en cuenta las características de la plataforma fue seleccionado dentro de la familia de estilos de Llamada y Retorno el estilo basado en componentes. Luego del estudio de los patrones de arquitectura, se observa que ninguno es aplicable a la arquitectura de la plataforma, ya que esta se centra en la integración de los diferentes componentes. Se propone aplicar los patrones de diseño GRASP. Como metodología de desarrollo se seleccionó OpenUP/Basic, utilizando UML para describir la arquitectura y como lenguaje de modelado, y Visual Paradigm como herramienta CASE. Se escoge como lenguaje de programación Java, utilizando Eclipse como entorno de desarrollo y Subversion como sistema de control de versiones. Se seleccionó la plataforma JEE y se recomienda hacer uso del framework Hibernate por los beneficios que presenta. El gestor de BD seleccionado fue PostgreSQL, y el servidor de aplicaciones Tomcat.

Capítulo

Descripción de la arquitectura

2.1 Introducción

A partir de la selección realizada en el capítulo anterior se presenta la aplicación de las metodologías, tecnologías y herramientas para el diseño de la arquitectura. En el presente capítulo se establecen las metas y restricciones que se deben cumplir para un mejor funcionamiento del sistema, así como la descripción del diseño de la arquitectura, ilustrando las diferentes vistas arquitectónicas.

2.2 Organización del sistema

En la Figura 2 se muestra la forma en que se propone organizar el sistema, para ofrecer una mejor idea y facilitar la comprensión de la arquitectura propuesta.

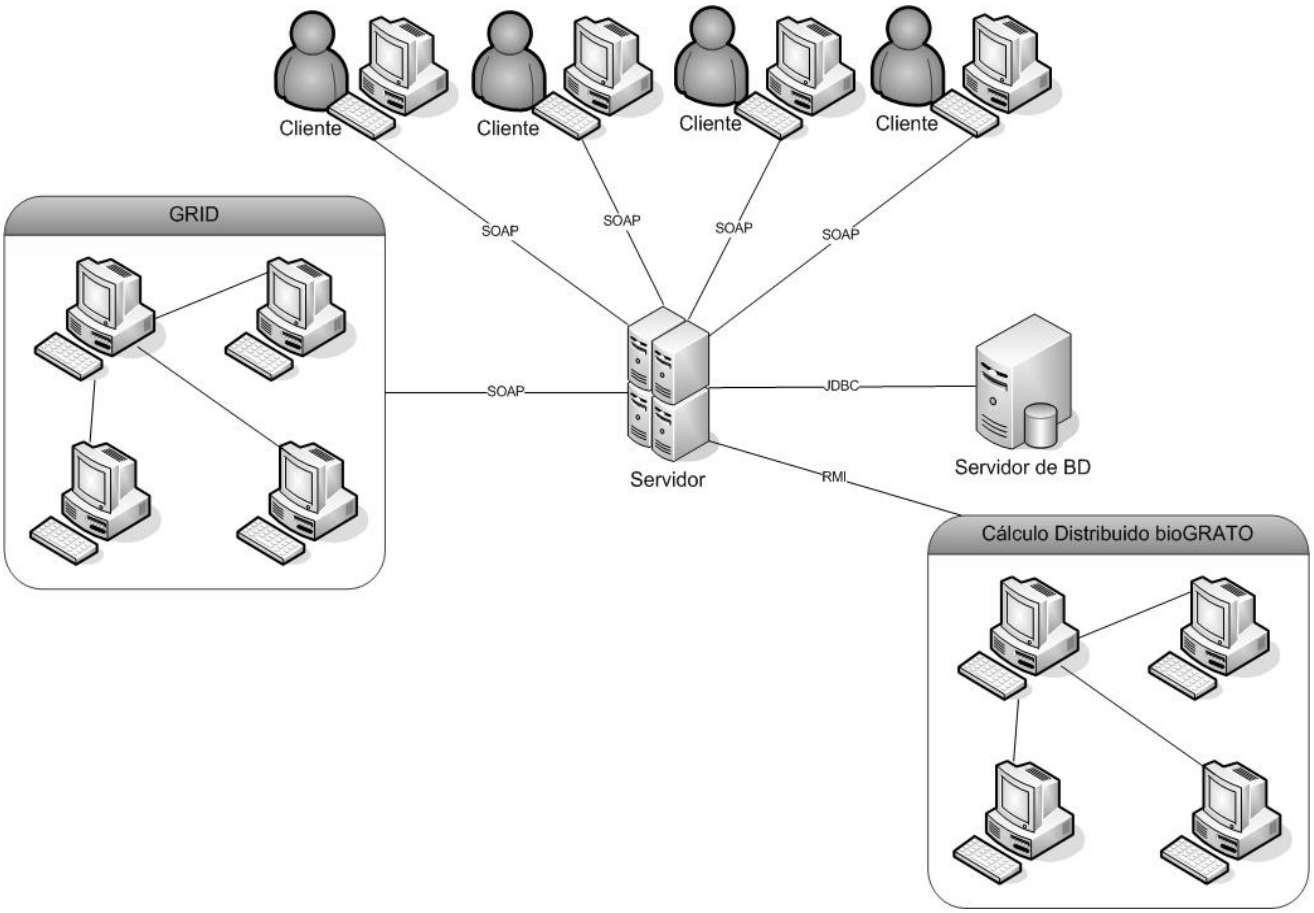

Figura 2. Organización del sistema.

Como se observa, el sistema contará con un servidor donde se coordinarán las peticiones realizadas por los clientes. Estas peticiones serán realizadas por las computadoras (PC) clientes a través del protocolo SOAP. Desde el servidor se realizará la conexión a la BD, extrayendo los datos para realizar el trabajo. Una vez obtenida la información, el servidor se encargará de enviar el trabajo al conjunto de PC que componen el cálculo distribuido bioGRATO. Concluidas estas operaciones, el servidor elabora la respuesta que es enviada a la PC del usuario, esta la visualiza y así termina el proceso.

En caso de que la petición realizada por el cliente requiera gran capacidad de cálculo, el trabajo se enviará a una GRID externa, donde se procesará la información.

Para un mejor entendimiento, se muestra y detalla cada una de las partes de la figura anterior:

Cliente

Como se ilustró en la figura, pueden existir varios clientes, que contarán con una aplicación de escritorio que permitirá visualizar proteínas y moléculas, así como las diferentes variaciones de estas.

Estas operaciones se realizarán a través de una aplicación de escritorio pues el trabajo de imágenes en la Web resulta lento y poco efectivo.

Servidor de BD

Contendrá la BD donde se encontrará información almacenada referente a las moléculas proteínas.

GRID

Constituirá una red de computadoras externa donde se podrán realizar operaciones complejas y que demanden gran capacidad de cálculo.

Servidor

Constituirá la parte fundamental dentro de la arquitectura, ya que aquí es donde se gestionan las solicitudes y respuestas de los clientes. El funcionamiento de este será como se muestra en la figura 3.

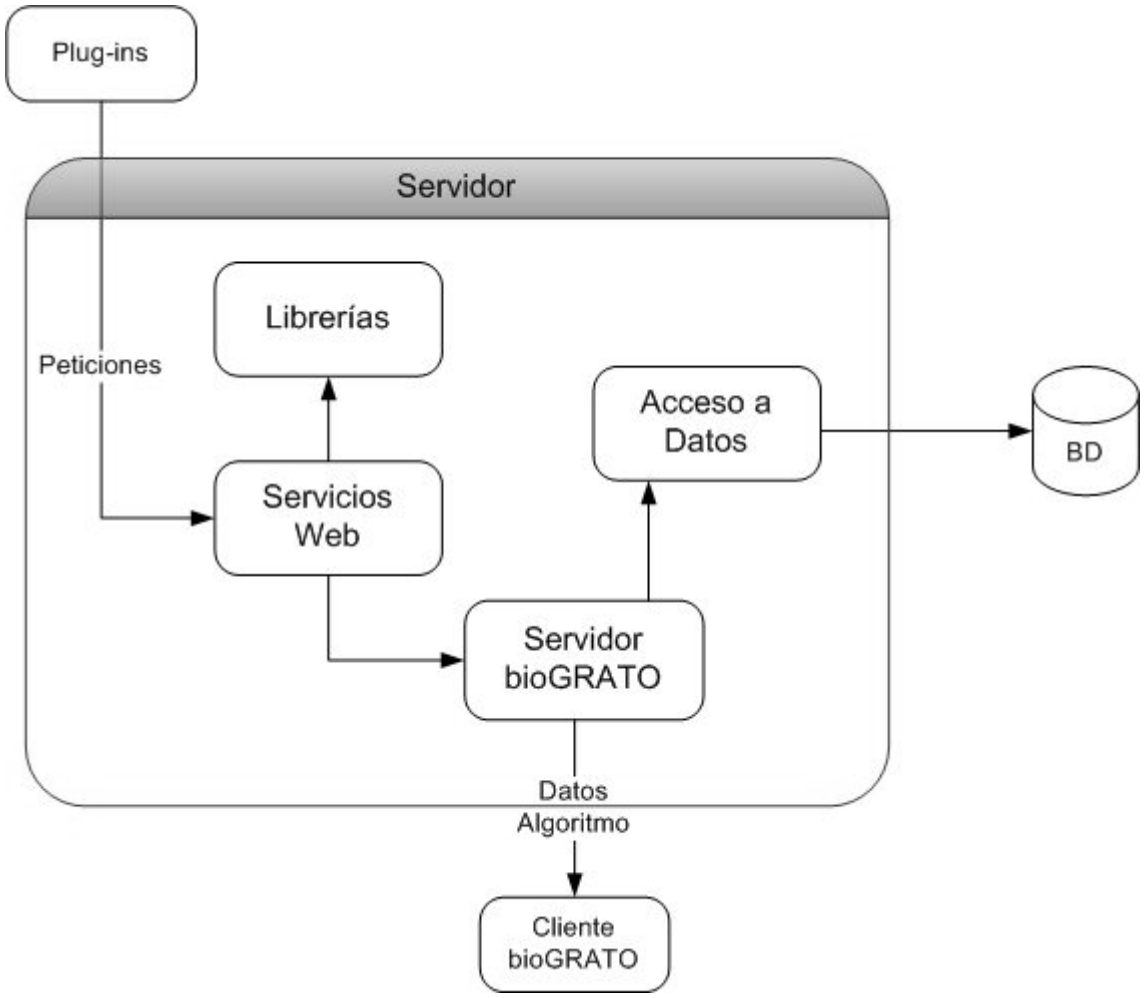

Figura 3. Funcionamiento en el Servidor.

Las peticiones de los usuarios serán manejadas a través de servicios Web. Estos servicios haciendo uso de librerías se conectan al Servidor bioGRATO enviándole las solicitudes de trabajo. En este servidor se gestiona la conexión a la BD, extrayendo los datos necesarios y distribuyendo el trabajo. Para realizar esta distribución, el Servidor bioGRATO envía los datos y el algoritmo por el cual debe ser resuelto el problema a cada una de las PCs distribuidas. Cuando estas máquinas obtienen la solución, la envían al servidor, quien se encarga de completar la respuesta, y a través de los servicios Web es enviada al usuario.

2.3 Metas y restricciones arquitectónicas

Los requerimientos no funcionales son propiedades o cualidades que el producto debe tener. Son las características que hacen al producto atractivo, usable, rápido o confiable.

Dentro de las metas y restricciones arquitectónicas para la plataforma, se encuentran los requisitos no funcionales, estos se enuncian a continuación.

Apariencia o interfaz externa

El sistema debe estar diseñado con una interfaz atractiva, fácil de usar, de forma tal que el usuario navegue sin dificultad alguna, ajustándose a los estándares establecidos para el desarrollo de un buen diseño.

Usabilidad

El sistema podrá ser usado por cualquier tipo de persona que posea conocimientos básicos en el manejo de la computadora, solo se necesita contar con conocimientos especializados en química para entender los resultados dados por la aplicación.

Soporte

El sistema debe propiciar su mejoramiento y la incorporación de otras opciones en el futuro.

Portabilidad

El sistema debe ser ejecutado sobre los sistemas operativos Linux y Windows, por su característica de ser multiplataforma.

Seguridad

El sistema debe establecer un mecanismo que permita la utilización de los servicios por usuarios autorizados. Debe contar con protección contra acciones no autorizadas o que puedan afectar la integridad de los datos.

Se deben garantizar comunicaciones seguras entre los clientes y el servidor.

Confiabilidad

El sistema debe ser confiable y preciso en la información que le suministra al usuario para evitar cualquier tipo de error.

Ayuda

El sistema debe poseer un material de asistencia al usuario, en la que se explique de forma clara el uso de las opciones del sistema garantizando así el buen desempeño de los usuarios a la hora de interactuar con el mismo.

Software

Se debe disponer de sistemas operativos Linux, Windows 95 o superior para la instalación de la aplicación

Debe tenerse instalado el Java Runtime Environment (JRE) versión 1.5 o superior.

Hardware

Para la puesta en práctica y desarrollo del proyecto se requieren máquinas con los siguientes requisitos:

Estaciones de Trabajo

Procesador Pentium 4 o superior.

256 MB de RAM.

40 GB de capacidad en disco.

Tarjeta de red.

Servidor de Aplicaciones

Procesador Pentium 4 a 3.00 GHz o superior.

1GB de memoria RAM.

40 GB de capacidad en disco.

Tarjeta de red.

Servidor de Base Datos

Procesador Pentium 4 a 3.00 GHz o superior.

1 GB de RAM.

160 GB de capacidad en disco.

Tarjeta de red.

PC Distribuida

Procesador Pentium 4 a 3.00 GHz o superior.

1 GB de RAM.

40 GB de capacidad en disco.

Tarjeta de red.

2.4 Vistas Arquitectónicas

La metodología utilizada, propone para describir la arquitectura, el modelo "4+1" de Philippe Kruchten, que define cuatro vistas diferentes de la arquitectura de software: (1) la vista lógica, que comprende las abstracciones fundamentales del sistema a partir del dominio del problema, (2) la vista de proceso: el conjunto de procesos de ejecución independiente a partir de las abstracciones anteriores, (3) la vista de despliegue: un mapeado del software sobre el hardware, (4) la vista de implementación: la organización estática de módulos en el entorno de desarrollo. El quinto elemento considera todos los anteriores en el contexto de casos de uso.[1]

El modelo 4+1 se percibe hoy como un intento de reformular una arquitectura estructural y descriptiva en términos de objeto y de UML. Con todo, las cuatro vistas de Kruchten forman parte del repertorio estándar de los practicantes de la disciplina. [1]

En el presente documento no se describirá la vista de proceso por no representar información relevante para la descripción de la arquitectura propuesta.

2.4.1 Vista de Casos de Uso

En esta vista se muestra la percepción que tiene el usuario de las funcionalidades del sistema mediante la representación de los actores y casos de usos más importantes. Si el sistema se hace extenso entonces se debe organizar en paquetes, lo cual facilitaría la comprensión de la vista.

Actores: Un actor representa un conjunto coherente de roles que los usuarios de casos de usos desempeñan cuando interaccionan con estos casos de uso.

Casos de Uso (CU): Los casos de uso describen un conjunto de secuencias de acciones, incluyendo variaciones que un sistema lleva a cabo y que conduce a un resultado observable de interés para un determinado actor.

Paquete: Un paquete es el elemento de organización básica de un modelo de sistema UML. Es un mecanismo general para dividir modelos y agrupar elementos.

Como se muestra en la figura 4, el sistema cuenta con dos actores, el Especialista y el Administrador. El Especialista representa el usuario que hará uso del sistema, teniendo la posibilidad de interactuar con todas la funcionalidades de este. El Administrador se encargará de realizar algunas tareas que se encuentran restringidas para el Especialista.

La vista se organizó en paquetes para simplificar su comprensión. En la elaboración de estos se tuvo en cuenta la estrecha relación entre los casos de uso, pues responden a funcionalidades altamente relacionadas.

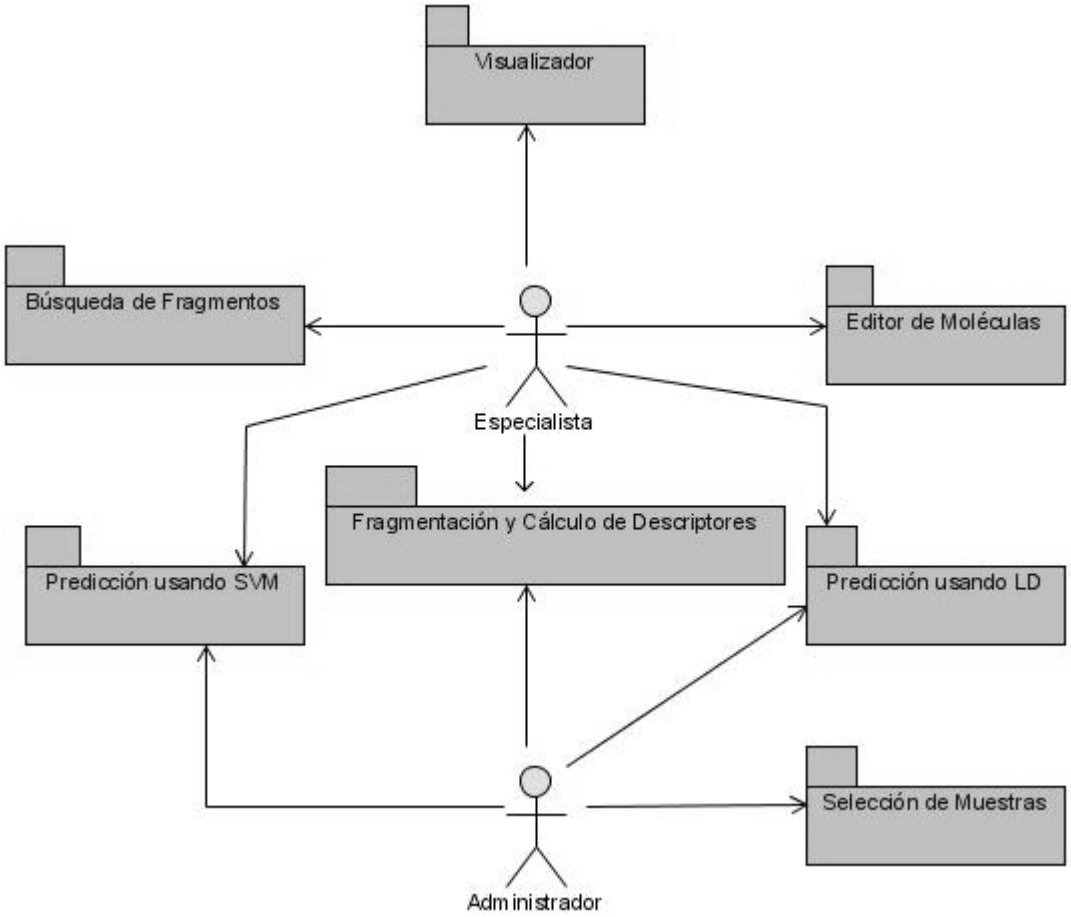

Figura 4. Vista de Casos de Uso.

A continuación se verán los casos de uso arquitectónicamente significativos correspondientes a cada paquete, los cuales fueron seleccionados porque el proceso que describen es de suma importancia para el sistema.

2.4.1.1 Paquete Búsqueda de Fragmentos

Buscar Fragmento: Permite al especialista cargar un fichero de extensión .mol en el editor y seleccionar los átomos que formarán el fragmento que él desea, el sistema le permitirá buscar en la base de datos, visualizando el resultado.

Gestionar Búsqueda: Permite al especialista filtrar y listar los resultados de la búsqueda. Mostrando las características específicas de cada uno de los resultados encontrados.

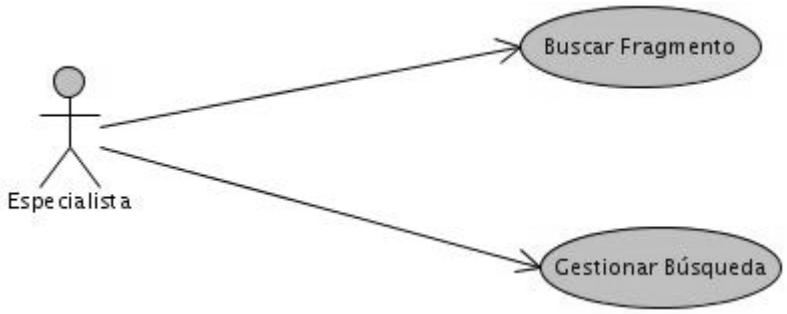

Figura 5. Diagrama CU del Paquete Búsqueda de Fragmentos.

2.4.1.2 Paquete Fragmentación y Cálculo de Descriptores

Calcular Descriptores por Molécula: Permite al especialista realizar la acción de calcular descriptores, para esto el sistema le brinda la posibilidad de cargar una molécula existente en un dispositivo, luego selecciona los descriptores a calcular, el sistema realiza la hibridación, fragmentación, y los cálculos de los descriptores a la molécula seleccionada, brindando la posibilidad de guardar esos resultados en un fichero.

Calcular Descriptores General: Permite al administrador realizar la acción de calcular descriptores de la base de datos.

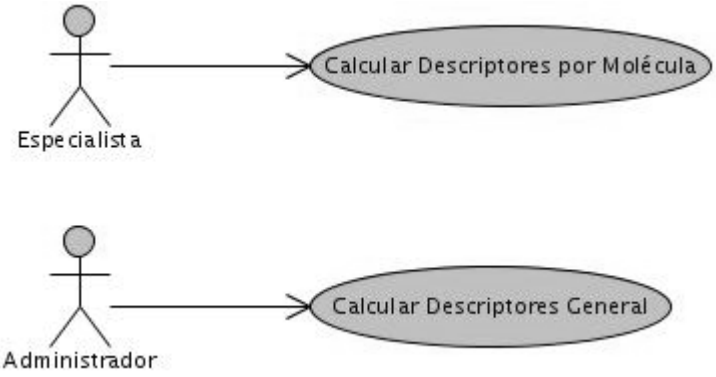

Figura 6. Diagrama de CU del Paquete Fragmentación y Cálculo de Descriptores.

2.4.1.3 Paquete Selección de Muestras

Realizar Selección: Permite al administrador introducir el fichero de entrada, el cual será reducido. El sistema comprueba que el fichero se encuentra en el formato correcto para poder realizar la reducción de la muestra. Comprobado lo anterior se realiza la selección de la nueva muestra, se convierte al formato de salida especificado por el usuario y se crea el fichero de salida.

Ordenar Selección: Permite al administrador introducir el fichero de entrada, el cual será reducido. El sistema comprueba que el fichero se encuentra en el formato correcto para poder realizar reducción de la muestra. Comprobado lo anterior se ordenan los descriptores o fragmentos moleculares con respecto a la actividad biológica, se convierte al formato de salida especificado por el usuario y se crea el fichero de salida.

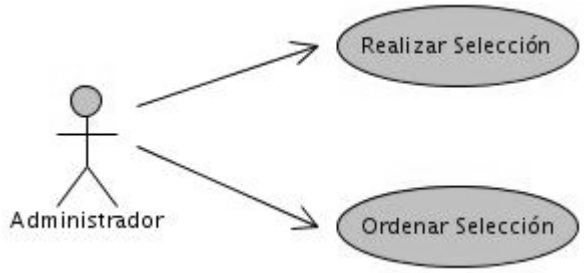

Figura 7. Diagrama de CU del Paquete Selección de Muestras.

2.4.1.4 Paquete Visualizador

Gestionar Plug-in: Permite al especialista adicionar o eliminar plug-ins.

Guardar Fichero: Permite al especialista convertir un fichero a la extensión que desee y guardarlo.

Calcular Propiedades: Permite al especialista calcular propiedades, como son la masa molecular, el por ciento de composición y la optimización geométrica.

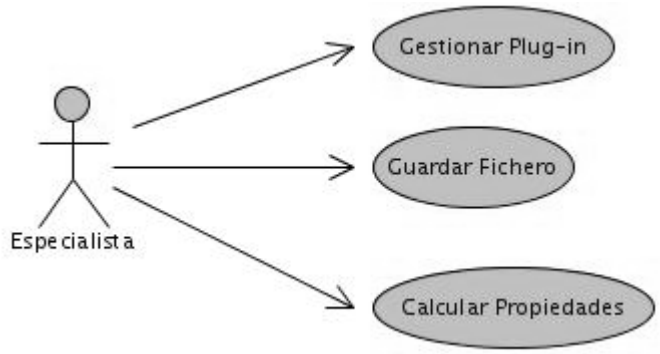

Figura 8. Diagrama de CU del Paquete Visualizador.

2.4.1.5 Paquete Editor de Moléculas

Crear Moléculas: Permite al especialista crear moléculas en el área de trabajo, brindándole varias opciones para el trabajo con esta, luego ofrece la opción de visualizarla.

Modificar Moléculas: Permite al especialista después de haber abierto una molécula modificar la estructura de la molécula que se encuentra seleccionada, almacenando los cambios en el fichero original, luego da la opción de visualizarla.

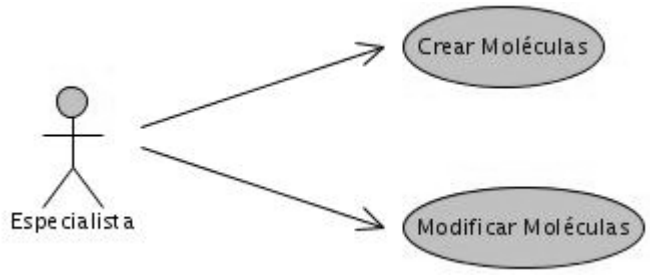

Figura 9. Diagrama de CU del Paquete Editor de Moléculas.

2.4.1.6 Paquete Predicción usando LD

Realizar Predicción: Permite al especialista realizar predicción, el especialista introduce sus moléculas y selecciona el modelo difuso de la base de datos. Luego con el proceso de defusificación se arriba a la predicción.

Crear Modelo Difuso: Permite al administrador crear modelo difuso. Luego puede trabajar con los ficheros de entrenamiento. Posteriormente se realiza el entrenamiento de los datos y se genera la base de conocimientos.

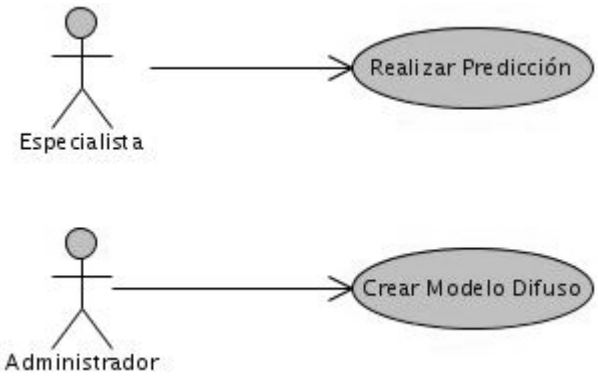

Figura 10. Diagrama de CU del Paquete Predicción usando LD.

2.4.1.7 Paquete Predicción usando SVM

Realizar Predicción: Permite al especialista cargar un fichero de prueba con la o las moléculas de las cuales desea saber su predicción, escoge un modelo que corresponda al entrenamiento realizado a ese tipo de datos, y asigna la dirección de un archivo vacío para la escritura de los resultados.

Crear Modelo: Permite al administrador seleccionar las opciones para realizar el entrenamiento, cargando el fichero con los datos de los fragmentos que presentan las actividades biológicas asociadas y el índice del estado refractotopológico total, el sistema realiza el entrenamiento y devuelve el modelo.

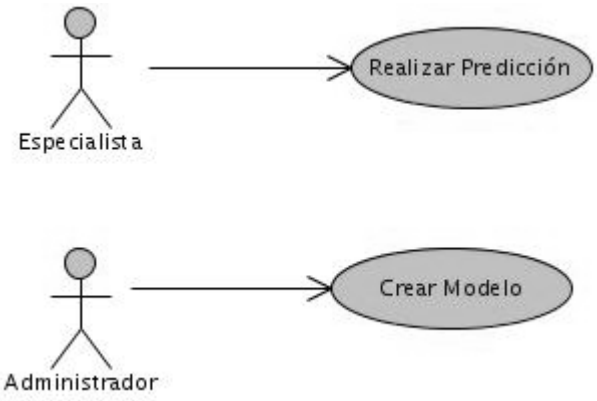

Figura 11. Diagrama de CU del Paquete Predicción usando SVM.

2.4.2 Vista Lógica

La vista Lógica representa los elementos de diseño más importantes para la arquitectura del sistema. Este describe las clases más importantes, su organización en paquetes y subsistemas.

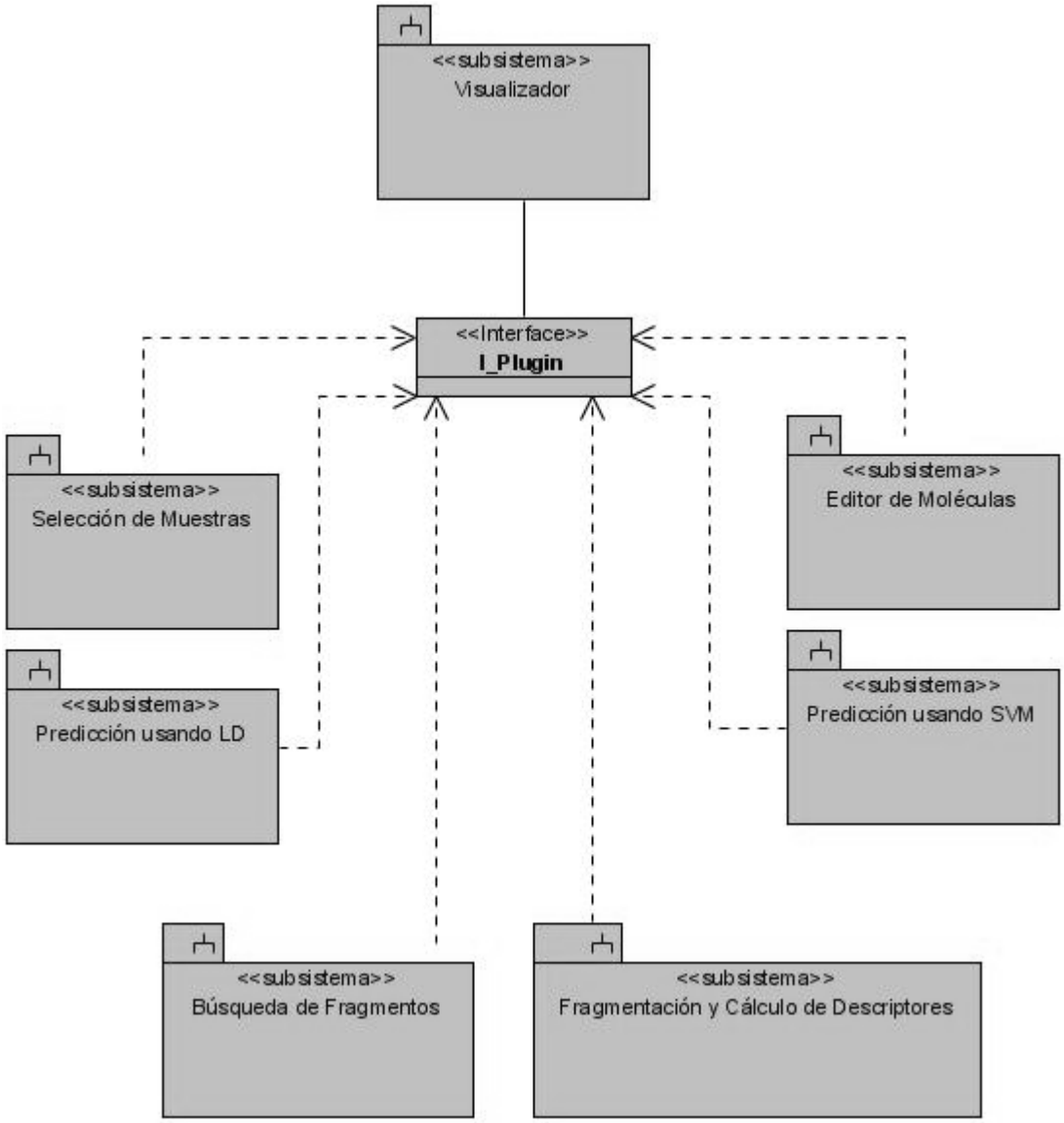

Figura 12. Vista Lógica.

Como se muestra en la figura 12 el sistema se estructuró por subsistemas. Un subsistema es la unidad de implementación compuesta por elementos funcionales, es decir, es la agrupación de partes funcionales del sistema. Un subsistema puede estar formado por un conjunto de clases y por otros subsistemas.

La organización en subsistemas se realiza porque hay módulos en los que existen muchas clases, y se hace casi imposible representarlas en un mismo diagrama, por eso se hace una organización de más alto nivel, que sería agrupar las clases en subsistemas. Entre los subsistemas se establecen relaciones, en dependencia de si un subsistema utiliza elementos de él mismo o de otro. Cada uno de

ellos puede brindar a los demás *interfaces* que estarán bien definidas por el arquitecto. Estas *interfaces* definirán operaciones que uno brindará y que otro u otros usarán. Este sistema está formado por siete subsistemas y una *interface*.

Visualizador

Este subsistema permite la visualización de compuestos químicos.

Subsistema Editor de Moléculas

Este subsistema se encarga de editar moléculas en 2d y 3d, facilitándole al químico la elaboración de nuevas moléculas con el fin de crear nuevos fármacos.

Subsistema Selección de Muestras

Este subsistema brinda una herramienta computacional capaz de reducir el espacio muestral de descriptores moleculares, para así realizar una mejor predicción de actividad biológica en la plataforma.

Subsistema Búsqueda de Fragmentos

Este subsistema permite analizar la similitud de un fragmento de una molécula con el resto de las estructuras almacenadas en la base de datos de la plataforma. El sistema devuelve las estructuras químicas encontradas en forma de reporte. Estas estructuras pueden ser seleccionadas para ver sus características físicas, además brinda la posibilidad de guardar el resultado en un fichero externo.

Subsistema Fragmentación y Cálculo de Descriptores

Este subsistema está centrado en el cálculo de descriptores atómicos y moleculares, para ello se realiza la hibridación molecular y fragmentación de una molécula guardando los resultados.

Subsistema Predicción usando LD

Este subsistema es el encargado de generar modelos predictivos de actividad biológica de compuestos orgánicos con la utilización de nuevas técnicas de Lógica Difusa.

Subsistema Predicción usando SVM

Este subsistema se encarga del procesamiento de compuestos moleculares orgánicos, a partir del análisis de su relación estructura-actividad biológica, utilizando para ello las Máquinas de Soporte Vectorial como técnica de Inteligencia Artificial.

Interface I_Plugin

Esta *interface* se encarga de contener operaciones para visualizar los resultados de cada uno de los subsistemas representados en la figura. Se propone implementar métodos que permitan obtener el nombre del plug-in, obtener la barra de herramientas contenidas en el plug-in y ejecutar la acción requerida.

2.4.3 Vista Despliegue

La vista de despliegue define la arquitectura física del sistema por medio de nodos interconectados. Estos nodos son elementos hardware sobre los cuales pueden ejecutarse los elementos software. El sistema estará distribuido como se muestra en la figura 13.

La aplicación será desplegada de la siguiente manera: la PC cliente, que podrá estar en cualquier parte del país, se conectará a un servidor de aplicaciones, mediante el protocolo SOAP, para hacer sus peticiones. Existirá un servidor de BD, que se encontrará conectado al servidor de aplicaciones por medio del *driver* JDBC, para realizar las consultas y accesos a la Base de Datos. El servidor será el encargado de distribuir las peticiones en 5 PC que estarán directamente conectadas a él, por medio del protocolo RMI, en las cuales se encontrarán todas las aplicaciones y funciones que soportará la plataforma. Cuando los cálculos sean demasiado grandes, se enviará el trabajo al servidor t-arenal externo a través del protocolo SOAP, quien se encargará de procesarlo y devolver la solución.

Inicialmente, se contará solo con 5 PC para realizar los cálculos, ya que este es el número mínimo para que el trabajo se realice correctamente, pero podrán adicionarse más, dependiendo de los recursos con los que se cuenten, para lograr una mejor eficiencia de la aplicación.

De esta forma, se garantizará el funcionamiento de la plataforma, y el despliegue de la misma en cualquier entorno investigativo que cuente con una red local o acceso a Internet.

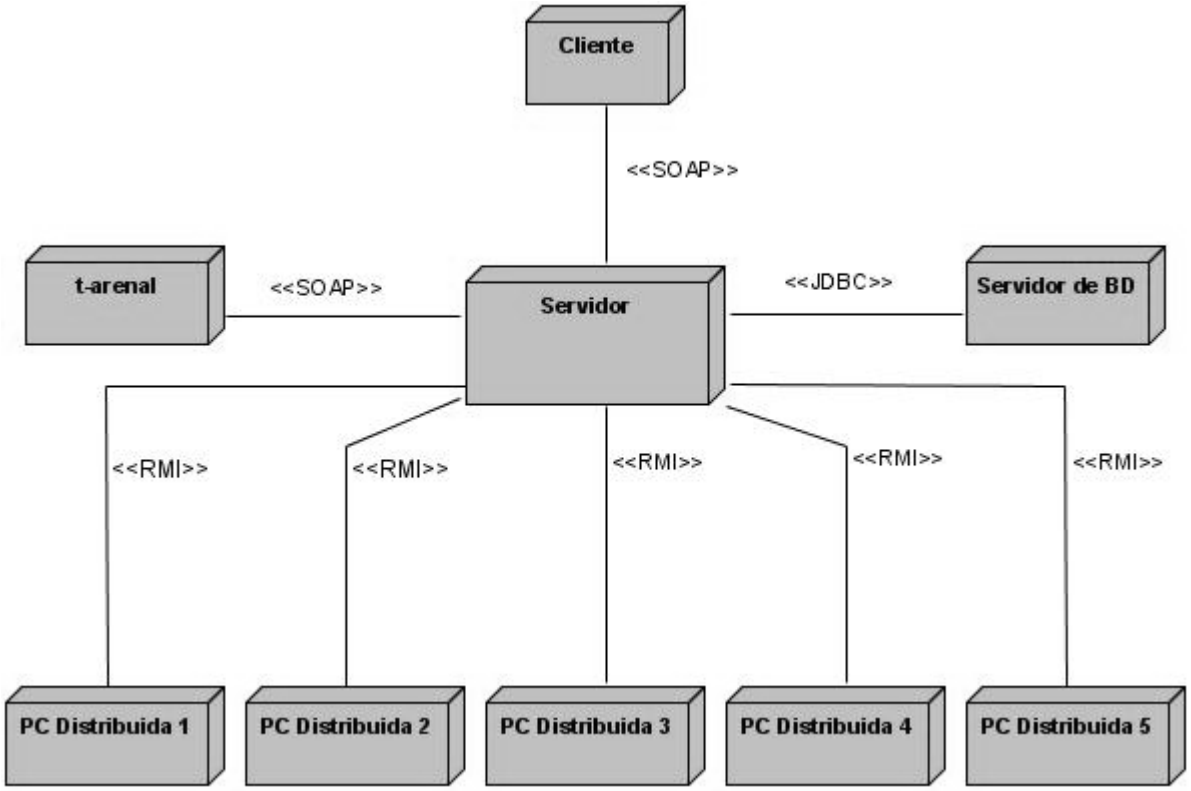

Figura 13. Vista de Despliegue.

2.4.4 Vista Implementación

La vista de implementación muestra el software como los elementos físicos que lo integran: componentes, ficheros, librerías. Como se muestra en la figura 14, el sistema se estructuró en componentes ejecutables y subsistemas de implementación.

Componente: Es la parte modular de un sistema, desplegable y reemplazable que encapsula implementación y un conjunto de interfaces y proporciona la realización de los mismos. Un componente típicamente contiene clases y puede ser implementado por uno o más artefactos (ficheros ejecutables, binarios).

Subsistema de Implementación: Una colección de componentes y otros subsistemas de implementación usados para estructurar el modelo de implementación y dividirlos en pequeñas partes.

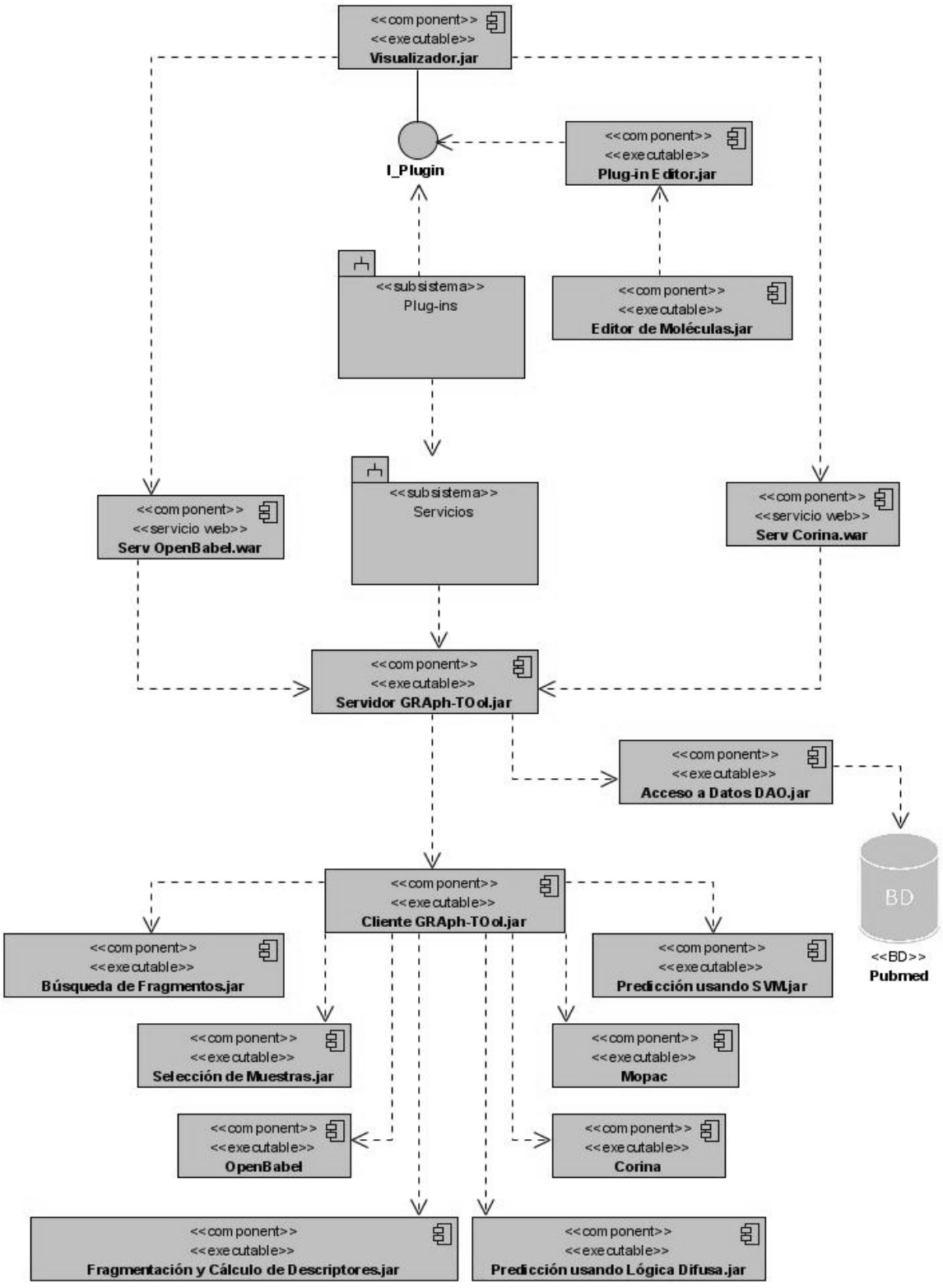

Figura 14. Diagrama de componentes.

Esta estructura posibilita la escalabilidad y la reusabilidad del sistema. Para la selección de los componentes se siguió el criterio de encapsulamiento por las funcionalidades de la plataforma. Todos estos componentes son importantes ya que representan una parte funcional del sistema. El componente Visualizador permite la gestión de las funcionalidades, así como brindar la *interface* de acceso a cada una de estas. Los componentes Editor de Moléculas, Búsqueda de Fragmentos, Fragmentación y Cálculo de Descriptores, Selección de Muestras, Predicción usando LD y Predicción usando SVM son los principales servicios que la plataforma brindará, estos pueden funcionar de forma independiente. El componente Acceso a Datos DAO gestionará la conexión a la base de datos quien almacenará y brindará la información.

Para facilitar la comprensión del diagrama de componentes se omitió la representación de la librería JdsGUIlibrary, pero vale destacar que esta será utilizada por los servicios Web para comunicarse con el Servidor bioGRATO. Esta librería se representa en la figura 17.

Como se muestra en la figura 15, el subsistema de implementación Servicios contiene los componentes Serv Búsqueda 2D y 3D, Serv Selección Muestras, Serv LD, Serv SVM y Serv Descriptores.

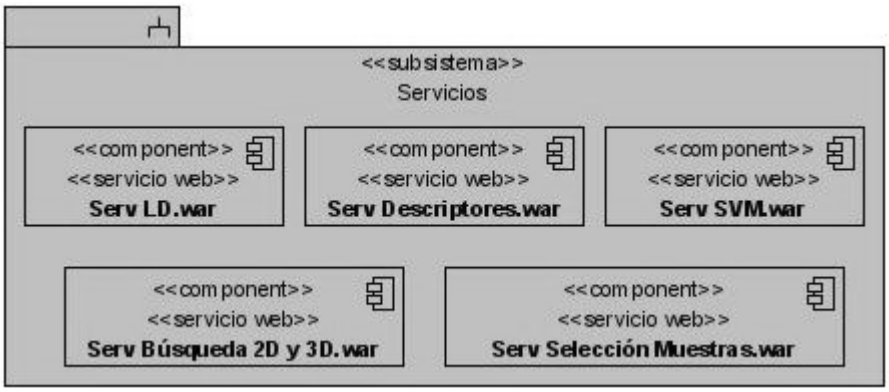

Figura 15. Subsistema Servicios

En la figura 16 se muestra el Subsistema Plug-ins, donde se encapsularon los componentes Plug-in Predicción LD, Plug-in Descriptores, Plug-in Predicción SVM, Plug-in Búsqueda y Plug-in Selección.

Cada plug-in de este subsistema se relacionará con un servicio Web del subsistema Servicios. El Plug-in Editor no se encuentra en el subsistema de la figura 15, por no tener relación con ningún servicio Web.

Los servicios Corina y OpenBabel, no se encuentran en el subsistema Servicios porque estos estarán conectados directamente al Visualizador sin tener asociado ningún plug-in.

Cada componente de tipo plug-in, deberá implementar la *Interface* que brinda el Visualizador.

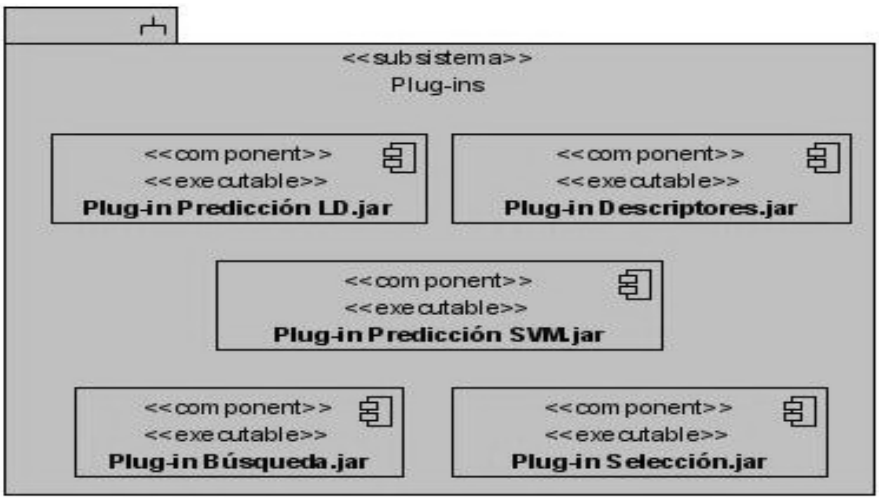

Figura 16. Subsistema Plug-ins

Los componentes Servidor y Cliente bioGRATO, son los encargados de gestionar la distribución de las operaciones que deben realizarse para obtener el resultado.

Distribución Física de los Componentes

En la figura 17 se muestra la distribución de los componentes en los nodos físicos. Además de los componentes desarrollados para la creación de la plataforma, se visualizan otros que son reutilizados y que constituyen programas independientes.

Los programas reutilizados son:

- Corina: Permite visualizar las moléculas en dos y tres dimensiones.
- OpenBabel: Permite hacer cambios entre diversos formatos.
- Mopac: Permite el cálculo de dinámica molecular.
- JdsGUILibrary: Permite la comunicación entre los servicios Web y el servidor bioGRATO.

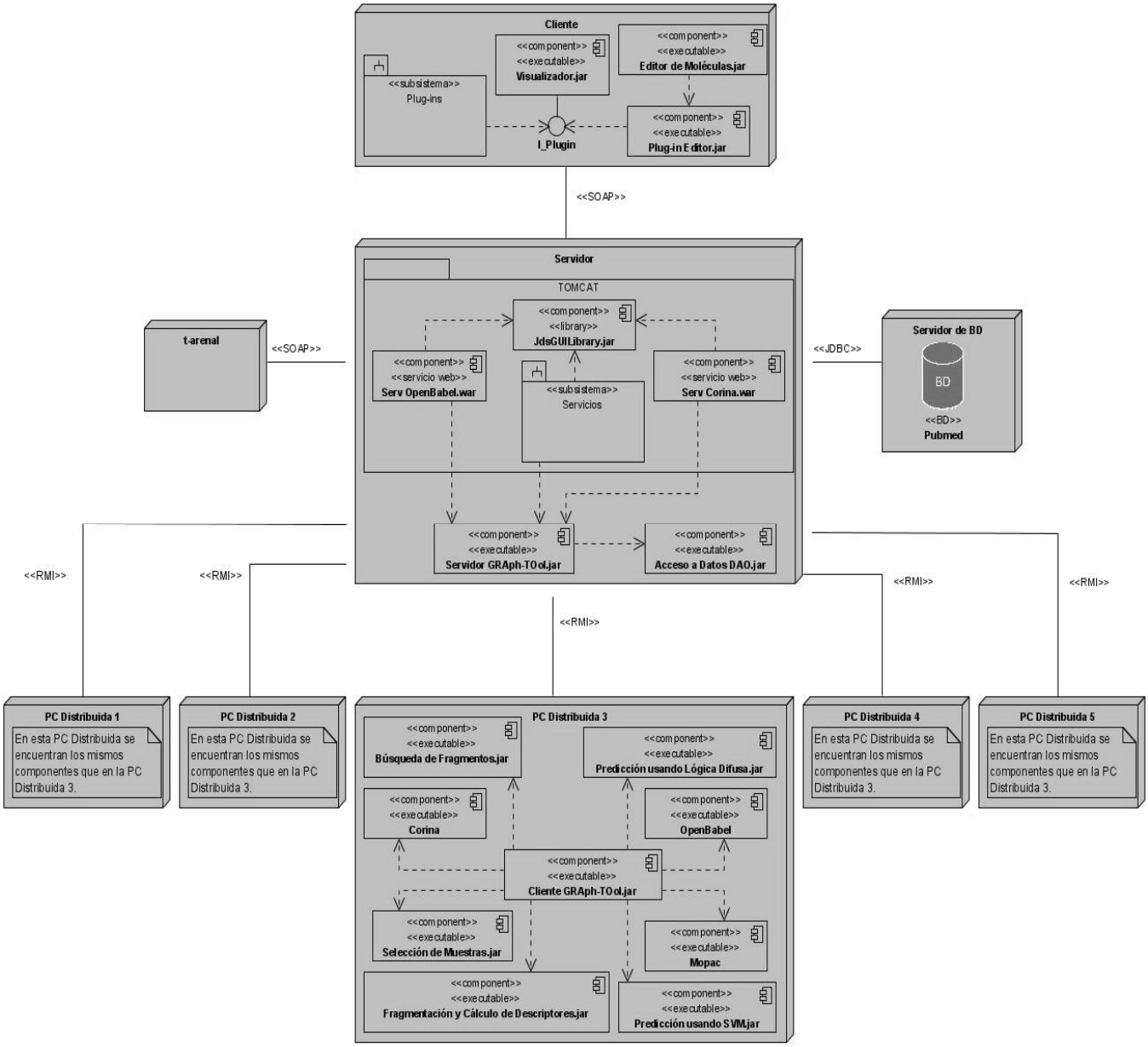

Figura 17. Distribución física de los componentes.

En el Nodo Cliente estarán los componentes Visualizador y Editor, ya que con estos puede trabajar el usuario estando en su puesto de trabajo sin hacer ninguna petición al servidor; además se encontrarán los plug-ins que brindarán la interfaz gráfica y la comunicación con los servicios Web; también se encontrará la *interface* I_Plugin. En el nodo Servidor estará instalado el servidor de aplicaciones Tomcat, donde se alojarán los servicios Web, que tendrá la plataforma y la librería JdsGUILibrary. En este nodo se encontrará, además, un componente de suma importancia como es el Servidor

bioGRATO que se encargará de distribuir el trabajo en las PCs Distribuidas para lograr un mejor rendimiento. En todas las PCs Distribuidas estarán todos los componentes de mayor peso: Editor de Moléculas, Búsqueda de Fragmentos, Fragmentación y Cálculo de Descriptores, Selección de Muestras, Predicción usando LD, Predicción usando SVM, Corina, OpenBabel y Mopac, ya que son los encargados de realizar los servicios que brinda la plataforma. En estas PCs estará ubicado el componente Cliente bioGRATO que será el responsable de recibir los trabajos enviados por el servidor y hacerlos ejecutar en los componentes antes mencionados.

En el nodo servidor de BD, se encontrará la base de datos de la plataforma. A esta base de datos se accederá a través del componente Acceso a Datos DAO que se encontrará en el nodo Servidor.

En caso de que la petición del cliente requiera de muchos cálculos y el componente de Servidor bioGRATO no cuente con los recursos necesarios para dar respuesta a dicha solicitud, se le envía el trabajo al Servidor T-Arenal para realizar la petición del usuario.

2.5 Conclusiones

En este capítulo se definió cual será la arquitectura que se utilizará en la Plataforma bioGRATO. Se dieron a conocer detalles de la organización que tendrá el sistema para un mejor entendimiento. Se plasmaron los requisitos no funcionales que harán que el funcionamiento de la plataforma sea el mejor posible. Se definieron y explicaron detalladamente las diferentes vistas arquitectónicas: vista de CU, vista Lógica, vista de Despliegue y vista de Implementación.

Capítulo

Evaluación del diseño arquitectónico propuesto

3.1 Evaluando la Arquitectura de Software

Evaluar una AS sirve para prevenir todos los posibles problemas de un diseño que no cumple con los requerimientos de calidad y para saber que tan adecuada es la AS diseñada para el sistema.

Una evaluación de una AS no te da un SI o un NO, si es buena o mala, o una calificación. Te dice donde está el riesgo, es decir fortalezas y debilidades identificadas de la AS.[35]

Razones para evaluar una Arquitectura de Software

Cuanto más temprano se encuentre un problema en un proyecto de software, mejor. El costo de arreglar un error durante las fases tempranas, es mucho menor al costo de arreglar ese mismo error en la fase de verificación. Dado que la arquitectura es un producto temprano de la fase de diseño, esta tiene un profundo efecto en el sistema y en el proyecto.

Una mala arquitectura puede llevar a un proyecto al fracaso. Todos los requerimientos de calidad pueden quedar insatisfechos.

La arquitectura también determina la estructura del proyecto: configuración, agenda y presupuesto, alcance, entre otros aspectos. Es mejor cambiar la arquitectura antes que otros artefactos, que están basados en ella, se estabilicen.

Realizar una evaluación de la arquitectura es la manera más económica de evitar desastres.

Momento para evaluar una Arquitectura

Generalmente, la evaluación de la arquitectura ocurre después que esta ha sido especificada, pero antes que empiece la implementación. En un proceso iterativo y/o incremental, la evaluación se puede realizar al final de cada ciclo. Sin embargo, uno de los atractivos de la evaluación de arquitecturas es

que se puede efectuar en cualquier etapa de la vida de una AS. En particular, existen dos variaciones útiles: temprana y tardía.

Evaluación temprana

La evaluación no tiene porque esperar a que la arquitectura este totalmente especificada. Esta puede ser utilizada en cualquier etapa del proceso de creación de la arquitectura, para examinar las decisiones arquitectónicas ya tomadas y decidir entre las opciones que están pendientes.

Por supuesto, la completitud y fidelidad de la evaluación es directamente proporcional a la completitud y fidelidad de la descripción de la arquitectura.

Evaluación tardía

Esta variación toma lugar no solo cuando la arquitectura está terminada, también cuando la implementación está completa. Este caso ocurre cuando la organización hereda un sistema legado. La técnica para evaluar un sistema legado es la misma que para evaluar un sistema recién nacido. Una evaluación es útil para entender el sistema legado, y saber si este cumple con los requerimientos de calidad y comportamiento.

En general, una evaluación debe realizarse cuando hay suficiente de la arquitectura como para justificarlo. Una buena regla sería: realizar una evaluación cuando el equipo de desarrollo empieza a tomar decisiones que dependen de la arquitectura y el costo de deshacerlas sobrepasa al costo de realizar una evaluación.

Involucrados en la evaluación de una AS

Hay dos grupos de personas involucrados en la evaluación de la arquitectura.

- Equipo de evaluación.
- *Stakeholders.*

Resultado que produce la evaluación de una Arquitectura

Un resultado que produce la evaluación de una arquitectura es la captura y priorización de las metas que la arquitectura debe cumplir para poder ser considerada adecuada.

La evaluación de una arquitectura no produce resultados cuantitativos. La evaluación ayuda a encontrar debilidades. No es de interés, por ejemplo, evaluar el desempeño en cantidad de transacciones por segundo en este momento, dado que el sistema no está construido aún; lo que

interesa, es un espíritu de mitigación de riesgos, es aprender cómo un atributo de calidad es afectado por una decisión de diseño arquitectónico, para que de esta manera se pueda estudiar con cuidado dicha decisión.

Una evaluación arquitectónica dice si una arquitectura es adecuada respecto a un conjunto de metas, y problemática con respecto a otro conjunto de metas. En ocasiones, las metas pueden ser contradictorias entre ellas, o algunas ser más importantes que otras. La evaluación ayuda a encontrar debilidades, no dirá “si” o “no”, “bien” o “mal”, “6 en 10”, dirá dónde están los riesgos.

Cualidades por las cuales puede ser evaluada una Arquitectura

No es del todo cierto que se pueda afirmar que el sistema alcanzará todas sus metas de calidad con solo mirar la arquitectura. Una arquitectura puede ser evaluada a través de atributos de calidad.

Los atributos de calidad se pueden clasificar en dos categorías:[8]

- **Observables vía ejecución:** aquellos atributos que se determinan del comportamiento del sistema en tiempo de ejecución.
- **No observables vía ejecución:** aquellos atributos que se establecen durante el desarrollo del sistema.

Dentro de los atributos de calidad observables vía de ejecución se pueden encontrar:

- **Confiabilidad:** Es la habilidad del sistema de continuar operando sobre el tiempo. Es usualmente medida en tiempo promedio entre fallas.
- **Disponibilidad:** Es la porción de tiempo en que el sistema está levantado y corriendo. Se mide como el tiempo transcurrido entre fallas, así como, cuán rápido el sistema está apto para reanudar y quedar operativo ante una falla.
- **Seguridad Interna:** Es la medida de la habilidad del sistema de resistirse al uso no autorizado y negar los servicios, mientras los provee a usuarios legítimos.
- **Seguridad Externa:** Ausencia de consecuencias catastróficas en el ambiente. Es la medida de ausencia de errores que generan pérdidas de información.
- **Funcionalidad:** Es la habilidad del sistema de hacer el trabajo para el cual fue construido.

Dentro de los atributos de calidad no observables vía de ejecución se pueden encontrar:

- **Modificabilidad:** Es la habilidad de realizar cambios al sistema en forma rápida y a bajo costo.

- **Portabilidad:** Es la habilidad del sistema de correr sobre diferentes ambientes. Estos ambientes pueden ser de hardware, de software o una combinación de ambos. Portabilidad es un caso particular de modificabilidad.
- **Variabilidad:** Es la capacidad de la arquitectura de ser expandida o modificada para producir nuevas arquitecturas. Variabilidad es importante cuando la arquitectura se va a utilizar como piedra fundamental de toda una familia de productos relacionados, como ser una línea de producto.
- **Integridad:** Es la ausencia de alteraciones inapropiadas de la información.
- **Configurabilidad:** Posibilidad que se otorga a un usuario experto a realizar ciertos cambios al sistema.
- **Escalabilidad:** Es el grado con el que se pueden ampliar el diseño arquitectónico, de datos o procedimental.
- **Mantenibilidad:** Es la capacidad de someter a un sistema a reparaciones y evolución. Capacidad de modificar el sistema de manera rápida y a bajo costo.
- **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.

Si otro atributo de calidad, además de los mencionados antes, es importante para determinado proyecto, este también debe formar parte de la evaluación.

Salidas de una evaluación arquitectónica

Las salidas de una evaluación arquitectónica son información e ideas sobre la arquitectura.

- Lista priorizada de los atributos de calidad requeridos.
- Riesgos y no riesgos.

Costos y beneficios de realizar una evaluación arquitectónica

El mayor beneficio que brinda la evaluación de una arquitectura, es que descubre los problemas que si se hubiesen dejado sin descubrir, habría sido mucho más costoso corregirlos luego. En breve, una evaluación produce una mejor arquitectura.

Algunos beneficios más que brinda la evaluación se presentan a continuación.

- Reúne a los *stakeholders.*
- Fuerza una articulación en las metas específicas de calidad.
- Fuerza una explicación clara de la arquitectura.
- Mejora la calidad de la documentación de la arquitectura.
- Descubre oportunidades de reuso.
- Resultan mejoras en las arquitecturas.

3.2 Métodos de Evaluación de Arquitecturas de Software

Hasta hace poco no existían métodos de utilidad general para evaluar arquitecturas de software. Si alguno existía, sus enfoques eran incompletos, y no repetibles, lo que no brindaba mucha confianza. En virtud de esto, múltiples métodos de evaluación han sido propuestos. A continuación se explican algunos de los más importantes.

3.2.1 Método de Análisis de Arquitecturas de Software

El Método de Análisis de Arquitecturas de Software (SAAM, del inglés *Software Architecture Analysis Method*) fue el primer método de evaluación basado en escenarios que surgió. El foco de este método es la modificabilidad.

Los creadores de SAAM idearon un método para evaluar, por medio de escenarios, los diferentes atributos de calidad que las arquitecturas de software demandaban. En la práctica SAAM ha demostrado ser útil para evaluar muchos atributos de calidad rápidamente, como portabilidad, modificabilidad, extensibilidad, integrabilidad, así como el cubrimiento funcional que tiene la arquitectura sobre los requerimientos del sistema. El método también puede ser utilizado para evaluar aspectos más ligados con la arquitectura como *performance* o confiabilidad.

SAAM puede ser utilizado para evaluar una o múltiples arquitecturas. Si se comparan dos o más se culmina el análisis con una tabla indicando las fortalezas y debilidades de cada una en cada escenario. Si se evalúa una sola, se culmina con un reporte señalando los componentes computacionales donde la arquitectura no alcanza el nivel requerido. En ningún caso se emite un valor absoluto acerca de la “calidad arquitectónica”.

El método de evaluación consiste en seis pasos que se detallan a continuación en la tabla 2.[8]

Tabla 2. Pasos contemplados por el método de evaluación SAAM

1. Desarrollo de escenarios.	Un escenario es una breve descripción de usos anticipados o deseados del sistema. De igual forma, estos pueden incluir cambios a escenarios los que puede estar expuesto el sistema en el futuro.
2. Descripción de la arquitectura.	La arquitectura (o las candidatas) debe ser descrita haciendo uso de alguna notación arquitectónica que sea común a todas las partes involucradas en el análisis. Deben incluirse los componentes de datos y conexiones relevantes, así como la descripción de la arquitectura comportamiento general del sistema. El desarrollo de escenarios y la descripción de la arquitectura son usualmente llevados a cabo de forma intercalada, o a través de varias iteraciones.
3. Clasificación y asignación de prioridad de los escenarios.	La clasificación de los escenarios puede hacerse en dos clases: directos e indirectos. Un escenario directo es el que puede satisfacerse sin la necesidad de modificaciones en la arquitectura. Un escenario indirecto es aquel que requiere modificaciones en la asignación de prioridad de la arquitectura para poder satisfacerse. Los escenarios indirectos son de especial interés para SAAM, pues son los que permiten medir el grado en el que una arquitectura puede ajustarse a los cambios de evolución que son importantes para los involucrados en el desarrollo.
4. Evaluación individual de los escenarios indirectos.	Para cada escenario indirecto, se listan los cambios necesarios sobre la arquitectura, y se calcula su costo. Una modificación sobre la arquitectura significa que debe introducirse un nuevo componente o conector, o que alguno de los existentes requiere cambios en su especificación.
5. Evaluación de la interacción entre	Cuando dos o más escenarios indirectos proponen cambios sobre un mismo componente, se dice que interactúan sobre ese componente. Es necesario evaluar este hecho, puesto que la interacción de la interacción componentes

escenarios.	semánticamente no relacionados revela que los componentes de la arquitectura efectúan funciones semánticamente distintas. De forma similar, puede verificarse si la arquitectura se encuentra documentada a un nivel correcto de descomposición estructural.
6. Creación de la evaluación global.	Debe asignársele un peso a cada escenario, en términos de su importancia relativa al éxito del sistema. Esta asignación de peso suele hacerse con base en las metas del negocio que cada escenario soporta. En el caso de la evaluación de múltiples arquitecturas, la asignación de pesos puede ser utilizada para la determinación de una escala general.

3.2.2 Método de Análisis de Acuerdos de Arquitectura

El Método de Análisis de Acuerdos de Arquitectura (ATAM, del inglés *Architecture Trade-off Analysis Method*) obtiene su nombre no solo porque nos dice cuan bien una arquitectura particular satisface las metas de calidad, sino que también provee ideas de cómo esas metas de calidad interactúan entre ellas, como realizan concesiones mutuas entre ellas.

Tener un método estructurado, permite hacer el análisis repetible y ayuda a asegurar que las preguntas acerca de una arquitectura serán contestadas en forma temprana, cuando es relativamente económico corregir problemas.

El ATAM también puede ser utilizado para analizar sistemas legados. Esta necesidad nace cuando el sistema legado necesita ser modificado, integrado con otro sistema, entre otras necesidades. Aplicar el ATAM incrementa el entendimiento de los atributos de calidad del sistema legado.

El método de evaluación ATAM comprende nueve pasos, agrupados en cuatro fases. A continuación, se muestran en forma de tablas las fases y sus pasos enumerados, junto con su descripción.

Tabla 3. Fase 1 método ATAM[8]

Fase 1: Presentación	
1. Presentación del ATAM.	El líder de evaluación describe el método a los participantes, trata de establecer las expectativas y responde las preguntas propuestas.
2. Presentación de las metas del negocio.	Se realiza la descripción de las metas del negocio que motivan el esfuerzo, y aclara que se persiguen objetivos de tipo arquitectónico.
3. Presentación de la arquitectura.	El arquitecto describe la arquitectura, enfocándose en cómo ésta cumple con los objetivos del negocio.

Tabla 4. Fase 2 método ATAM[8]

Fase 2: Investigación y análisis	
4. Identificación de los enfoques arquitectónicos.	Estos elementos son detectados, pero no analizados.
5. Generación del *Utility Tree.*	Se listan los atributos de calidad que engloban la “utilidad” del sistema (desempeño, disponibilidad, seguridad, modificabilidad, usabilidad), especificados en forma de escenarios. Se anotan los estímulos y respuestas, así como se establece la prioridad entre ellos.
6. Análisis de los enfoques arquitectónicos.	Con base en los resultados del establecimiento de prioridades del paso anterior, se analizan los elementos del paso 4. En este paso se identifican riesgos arquitectónicos, puntos de sensibilidad y puntos de balance.

Tabla 5. Fase 3 método ATAM[8]

Fase 3: Pruebas	
7. Lluvia de ideas y establecimiento de prioridad de escenarios.	Con la colaboración de todos los involucrados, se complementa el conjunto de escenarios.
8. Análisis de los enfoques arquitectónicos.	Este paso repite las actividades del paso 6, haciendo uso de los resultados del paso 7. Los escenarios son considerados como casos de prueba para confirmar el análisis realizado hasta el momento.

Tabla 6. Fase 4 método ATAM[8]

Fase 4: Reporte	
9. Presentación de los resultados.	Basado en la información recolectada a lo largo de la evaluación del ATAM, se presentan los hallazgos a los participantes.

3.2.3 Método de Revisiones Activas para Diseños Intermedios

Las arquitecturas generalmente crecen lentamente, a través de una serie de pasos y cada uno de estos pasos presentan complejos problemas de subdiseño. Si estos diseños intermedios se resuelven de una manera inapropiada, la arquitectura puede no ser adecuada para el proyecto.

Por esto hacerle revisiones a la arquitectura en las etapas intermedias nos provee una valiosa visón de la viabilidad de la arquitectura a construir, así como también nos permite descubrir errores e inconsistencias. En un futuro la mayoría de los proyectos van a realizar estas revisiones en sus subsistemas.

Lo que se requiere en estas etapas, es una liviana evaluación que se concentre en la conveniencia, expone el diseño a los interesados provocando interés y pueda ser llevada a cabo en la ausencia de una detallada documentación. El método de Revisiones Activas para Diseños Intermedios (ARID, del inglés *Active Reviews for Intermediate Designs*) cumple con todas estas características y se basa en

las mejores cualidades de los métodos basados en escenarios (ATAM o SAAM) y las revisiones activas de diseños (ADRS).

El método ARID es conveniente para realizar la evaluación de diseños parciales en las etapas tempranas del desarrollo.

Las revisiones de diseño activas y los ATAM tienen características útiles para resolver el problema de evaluar diseños preliminares. ARID surge de combinar las mejores cualidades de los métodos ADRs y los métodos basados en escenarios.

Las Tablas 7 y 8 presentan las fases y los pasos que involucra el método de evaluación ARID, con una breve descripción de cada uno.

Tabla 7. Fase 1 método ARID[8]

Fase 1: Actividades Previas	
1. Identificación de los encargados de la revisión.	Los encargados de la revisión son los ingenieros de software que se espera que usen el diseño, y todos los involucrados en el diseño. En este punto, converge el concepto de encargado de revisión de ADR e involucrado del ATAM.
2. Preparar el informe de diseño.	El diseñador prepara un informe que explica el diseño. Se incluyen ejemplos del uso del mismo para la resolución de problemas reales. Esto permite al facilitador anticipar el tipo de preguntas posibles, así como identificar áreas en las que la presentación puede ser mejorada.
3. Preparar los escenarios base.	El diseñador y el facilitador preparan un conjunto de escenarios base. De forma similar a los escenarios del ATAM y el SAAM, se diseñan para ilustrar el concepto de escenario, que pueden o no ser utilizados para efectos de la evaluación.
4. Preparar los materiales.	Se reproducen los materiales preparados para ser presentados en la segunda fase. Se establece la reunión, y los involucrados son invitados.

Tabla 8. Fase 2 método ARID[8]

Fase 2: Revisión	
5. Presentación del ARID.	Se explica los pasos del ARID a los participantes.
6. Presentación del diseño.	El líder del equipo de diseño realiza una presentación, con ejemplos incluidos. Se propone evitar preguntas que conciernen a la implementación o argumentación, así como alternativas de diseño. El objetivo es verificar que el diseño es conveniente.
7. Lluvia de ideas y establecimiento de prioridad de escenarios.	Se establece una sesión para la lluvia de ideas sobre los escenarios y el establecimiento de prioridad de escenarios. Los involucrados proponen escenarios a ser usados en el diseño para resolver problemas que esperan encontrar. Luego, los escenarios son sometidos a votación, y se utilizan los que resultan ganadores para hacer pruebas sobre el diseño.
8. Aplicación de los escenarios.	Comenzando con el escenario que contó con más votos, el facilitador solicita el seudo-código que utiliza el diseño para proveer el servicio, y el diseñador no debe ayudar en esta tarea. Este paso continúa hasta que ocurra alguno de los siguientes eventos: - Se agota el tiempo destinado a la revisión. - Se han estudiado los escenarios de más alta prioridad. - El grupo se siente satisfecho con la conclusión alcanzada. Puede suceder que el diseño presentado sea conveniente, con la exitosa aplicación de los escenarios, o por el contrario, no conveniente, cuando el grupo encuentra problemas o deficiencias.
9. Resumen.	Al final, el facilitador recuenta la lista de puntos tratados, pide opiniones de los participantes sobre la eficiencia del ejercicio de revisión, y agradece por su participación.

3.3 Evaluando la arquitectura de la Plataforma bioGRATO

Para la evaluación de la arquitectura propuesta, se utilizó el método ARID, por proporcionar la evaluación en fases tempranas del diseño, permitiendo así realizar cambios cuando estos no demanden demasiados esfuerzos. ARID constituye un método conveniente para la evolución de diseños parciales en las etapas tempranas del desarrollo y la técnica de evaluación basada en escenario.

Con el objetivo de obtener la evaluación se seleccionaron algunos atributos de calidad y un listado de riesgos.

Los atributos seleccionados fueron:

- Escalabilidad.
- Portabilidad.
- Mantenibilidad.
- Configurabilidad.
- Integridad.
- Seguridad Interna.

Los riesgos asociados a los atributos seleccionados son:

- Añadir servicios a la plataforma.
- Cambio de sistema operativo.
- Migración de sistema gestor de BD
- Mantenimiento a la plataforma.
- Adicionar funciones a la plataforma.
- Acceso a los datos.
- Intento de acceso no permitido.

A continuación, se plantea de qué forma se tratan estos riesgos atendiendo a los diferentes atributos de calidad seleccionados.

Tabla 9. Atributo Escalabilidad.

Atributo de calidad	**Perfil**	**Escenario**
Escalabilidad	Escalabilidad	Añadir servicios a la plataforma
Relación atributo – escenario		
La característica fundamental de la arquitectura es su escalabilidad. Lo servicios que se brindan a la comunidad química para su investigación, fueron programados como componentes independientes, permitiendo que estos puedan ser añadidos con facilidad. La definición de un nuevo módulo se puede hacer fácil y rápidamente, todo dependiendo del cálculo o proceso químico que se desee implementar. El proceso de incorporación de un nuevo módulo a la plataforma se realizaría con la implementación de un nuevo componente compuesto por un servicio Web y una aplicación que permita realizar la operación que se desea, además en esta implementación se seguiría una serie de pasos bien definidos, heredando de clases existentes, implementando interfaces y definiendo el plug-in que permitirá la creación de la interfaz visual del nuevo servicio.		

Tabla 10. Atributo Portabilidad.

Atributo de calidad	**Perfil**	**Escenario**
Portabilidad	Portabilidad	Cambio de sistema operativo
Relación atributo – escenario		
Las decisiones arquitectónicas tienen gran influencia en la portabilidad de un producto de software. La primera decisión importante en aras de obtener un producto portable fue al escoger las herramientas y tecnologías para el desarrollo de la Plataforma. El que la plataforma sea implementada en Java, y que use como gestor de BD PostgreSQL, permite que esta corra tanto sobre el sistema operativo Linux como Windows, o sobre cualquier sistema operativo que posea una máquina virtual de Java.		

Tabla 11. Atributo Mantenibilidad.

Atributo de calidad	**Perfil**	**Escenario**
Mantenibilidad	Mantenimiento	Migración de sistema gestor de BD
Relación atributo – escenario		
Para el escenario de migración de sistema gestor de bases de datos, por una necesidad de los desarrolladores o incluso de los clientes, al desarrollar la plataforma usando como framework Hibernate, no existirán graves problemas. Este framework es una clara implementación del patrón DAO, pues crea una capa separada que se ocupa del acceso a datos con total independencia del gestor y BD, dando la oportunidad de trabajar con varios gestores y bases de datos dentro de la misma aplicación sin que esto cree ningún conflicto en el modelo de objetos. Esto posibilita realizar de forma sencilla el cambio de SGBD, incluso, la existencia de varios de ellos.		

Tabla 12. Atributo Mantenibilidad.

Atributo de calidad	**Perfil**	**Escenario**
Mantenibilidad	Mantenimiento	Mantenimiento a la Plataforma
Relación atributo – escenario		
Al definir una arquitectura basada en componentes, se simplifica el mantenimiento del sistema, pues se puede retirar un componente para repararlo sin necesidad de que la plataforma deje de prestar sus servicios. Además, una vez reparado el componente, se le pueden aplicar pruebas de manera independiente, sin integrarlo a los demás, para garantizar su correcto funcionamiento. Dado que un componente puede ser mejorado continuamente, la calidad de la plataforma mejorará con el paso del tiempo.		

Tabla 13. Atributo Configurabilidad.

Atributo de calidad	**Perfil**	**Escenario**
Configurabilidad	Configuración	Adicionar funcionalidades a la plataforma
Relación atributo – escenario		
Al hablar de configurabilidad, se hace referencia a la posibilidad que tiene el usuario de realizar cambios en el sistema, de manera que pueda adaptarlo a sus necesidades. En este sentido, el usuario es capaz de incorporar o quitar funcionalidades al sistema según la acción que desee hacer. El será el responsable de configurar con que servicios desea trabajar y con cuales no, si posteriormente, desease cambiar de nuevo, podría hacerlo. Cuenta con la posibilidad de agregar o quitar los siguientes servicios: fragmentación y cálculo de descriptores, predicción de actividad biológica por LD o SVM, búsqueda de fragmentos y selección de muestras.		

Tabla 14. Atributo Integridad.

Atributo de calidad	**Perfil**	**Escenario**
Integridad	Integridad	Acceso a los datos
Relación atributo – escenario		
La integridad de la Información resulta extremadamente importante en los sistemas informáticos. Para mantener la integridad de la información en la Plataforma bioGRATO, se han definido varias estrategias. Se restringe según el rol y los servicios instalados al usuario, el acceso a la base de datos, y en esta se guarda la información que se refiere al tipo de servicio específico, permitiendo así no tener redundancia. Solo el rol de Administrador será el responsable de modificar los datos, mientras el Especialista solo podrá consultarlos. De esta forma, y contando con un buen diseño en la base de datos, se asegura la integridad de la información.		

Tabla 15. Atributo Seguridad Interna.

Atributo de calidad	**Perfil**	**Escenario**
Seguridad Interna	Seguridad	Intento de acceso no permitido
Relación atributo – escenario		
Para el uso de los servicios de la plataforma, se propone establecer un método de firma digital. El uso de este método, permite que solo usuarios autorizados puedan hacer uso de las funcionalidades que brinda la plataforma. A través de la negociación de llaves públicas y privadas, se conocerá que usuario está autorizado y cual no, denegando el acceso a quienes no posean la autorización requerida. Además, como mecanismo de seguridad adicional, ningún usuario podrá insertar datos en la BD, sólo el administrador será el encargado de esta operación, manteniendo así la seguridad sobre los datos.		

3.4 Conclusiones

En este capítulo se analizó la importancia de evaluar la arquitectura de software, y se analizaron además, los métodos para la evaluación arquitectónica. Se evaluó el diseño arquitectónico propuesto mediante el método ARID, permitiendo así, tener una evaluación de la arquitectura en una etapa temprana del diseño, mostrando como se puede analizar el cumplimiento de los atributos de calidad atendiendo a cada una de las vistas desarrolladas.

Conclusiones generales

La investigación realizada mostró que no existe un modelo unificado en cuanto a la definición, clasificación y aplicación de los elementos esenciales de la arquitectura de software. Sin embargo, prevalece el criterio de que la AS permite organizar el proceso de desarrollo y definir la estructura que tendrá el sistema.

Como resultado se obtuvo el diseño de la arquitectura de la Plataforma Inteligente para la Predicción de Actividad Biológica de Compuestos Orgánicos. Se utilizó el estilo Basado en Componentes, y se observó que ninguno de los patrones arquitectónicos analizados era aplicable al diseño propuesto. Se diseñaron las vistas del sistema utilizando el modelo de las 4+1 de Philippe Kruchten, de las cuales se representaron la vista de CU, la vista Lógica, la vista de Despliegue y la vista de Implementación. Además se describió la arquitectura, permitiendo el entendimiento y comprensión de la misma por parte del equipo de desarrollo. Contribuyó en la obtención del resultado alcanzado, la evaluación del diseño arquitectónico propuesto, la cual se realizó a través del método ARID, donde se evidenció de qué forma se cumplieron los atributos de calidad.

Recomendaciones

Luego de haber analizado los resultados del presente trabajo de diploma, resulta factible arribar a las siguientes recomendaciones:

- Poner en práctica el diseño arquitectónico propuesto.
- Refinar la arquitectura propuesta a partir de la puesta en práctica del diseño realizado.
- Valorar por parte de la Universidad de las Ciencias Informáticas la utilización del diseño arquitectónico propuesto para el desarrollo de nuevas aplicaciones informáticas con características similares.

Referencias bibliográficas

1. Reynoso, C.B. *Introducción a la Arquitectura de Software*. 2004 [cited 2007 noviembre]; Available from: http://www.willydev.net/descargas/prev/IntroArq.pdf.

2. Jason Baragry, K.R. *Why We Need a Different View of Software Architecture*. 2001 [cited 2007 noviembre]; Available from: http://www.eng.auburn.edu/csse/classes/comp7700/resources/Why_We_Need_A_Different_View_of_Software_Architecture.pdf.

3. Clements, P.C. *Coming Attractions in Software Architecture*. 1996 [cited 2007 noviembre]; Available from: http://www.sei.cmu.edu/pub/documents/96.reports/pdf/tr008.96.pdf.

4. Erich Gamma, R.H., Ralph Johnson, John Vlissides. *Design Patterns: Elements of reusable object-oriented software*. 1995 [cited 2007 noviembre]; Available from: http://www.uml.org.cn/c++/pdf/DesignPatterns.pdf.

5. Pressman, R.S., *Ingienería del Software. Un enfoque práctico.* V ed. Vol. I. 2005.

6. Len Bass, P.C., Rick Kazman, *Software Architecture in Practice*. 2003: Addison-Wesley Professional. 560.

7. Kruchten, P. *Architectural Blueprints—The "4+1" View Model of Software Architecture*. 1995 [cited 2007 noviembre]; Available from: http://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf.

8. CAMACHO, E., F. CARDESO, and G. NUÑEZ. *Arquitecturas de Software*. 2004 [cited 2007 noviembre]; Available from: http://prof.usb.ve/lmendoza/Documentos/PS-6116/Guia%20Arquitectura%20v.2.pdf.

9. Carlos Reynoso, N.K. *Estilos y Patrones en la Estrategia de*

Arquitectura de Microsoft. 2004 [cited 2007 noviembre]; Available from: http://www.willydev.net/descargas/prev/Estiloypatron.pdf.

10. Yorio, D. *Identificación y Clasificación de Patrones en el Diseño de Aplicaciones Móviles.* 2006 [cited 2007 noviembre 2007]; Available from:

http://postgrado.info.unlp.edu.ar/Carrera/Magister/Ingenieria%20de%20Software/Tesis/Yorio.pdf
.

11. Rosanigo, Z.B. *Modelos y Patrones*. 2000 [cited 2007 diciembre]; Available from: http://postgrado.info.unlp.edu.ar/Carrera/Magister/Ingenieria%20de%20Software/Tesis/Rosanigo.pdf.

12. Welicki, L. *Patrones y Antipatrones: una Introducción* 2007 [cited 2007 diciembre]; Available from: http://www.microsoft.com/spanish/msdn/comunidad/mtj.net/voices/MTJ_3317/default.aspx#M15
.

13. Larman, C., *UML y PATRONES*. 2004 ed. 1999.

14. Carlos Reynoso, N.K. *Lenguajes de Descripción de Arquitectura (ADL)*. 2006 [cited 2007 diciembre]; Available from: http://www.microsoft.com/spanish/msdn/arquitectura/roadmap_arq/lenguaje.mspx.

15. Nicolás Kicillof, D.Y. *Detecting and Solving Architectural Problems with JACAL*. 2004 [cited 2007 diciembre]; Available from: http://www-2.dc.uba.ar/profesores/nicok/Jacal1%20(Trans).pdf.

16. Robles, L.A. *Sistema Seguidor de Objetos (1)*. 2003 [cited 2007 diciembre]; Available from: http://ccc.inaoep.mx/~labvision/doo/proy/T72.pdf.

17. Sánchez, D.M. *Técnicas y Metodologías en el desarrollo de Software*. 2007 [cited 2007 diciembre]; Available from: http://kybele.escet.urjc.es/documentos/HC/HC4GL2007-T2-TecnicasI.pdf.

18. Ivar Jacobson, G.B., James Rumbaugh, *El Proceso Unificado de Desarrollo*, ed. A. Wesley. 2000, Madrid. 440.

19. Périssé, M.C., *Una Metodología Simplificada*. 2001, Buenos Aires. 203.

20. Autores, C.d. *Rational Rose 2000e Using Rose*. 2000 [cited 2007 diciembre]; Available from: http://www.vico.org/aRecursos/Rational/Rose_usingrose.pdf.

21. Seco, J.A.G., *El lenguaje de programación C#*. 2001, Barcelona. 301.

22. Ascui, I.C. *Java y la Programación Orientada a Objetos*. 2006 [cited 2007 diciembre]; Available from: http://iverclaros.blog.galeon.com/1141775160/.

23. Salazar, J.S. *Curso de Java*. 2005 [cited 2007 diciembre]; Available from: http://tikal.cifn.unam.mx/%7Ejsegura/LCGII/java3.htm.

24. García, L. *Sistema de control de versiones: SUBVERSION* 2008 [cited 2008 enero]; Available from: http://observatorio.cnice.mec.es/modules.php?op=modload&name=News&file=article&sid=548.

25. Retamar, A. *Mi primera hora con Eclipse.* 2004 [cited 2007 diciembre]; Available from: http://asturlinux.org/archivos/jornadas2004/ponencias/eclipse_ide.pdf.

26. Ted Husted, C.D., George Franciscus, David Winterfeldt, *Struts in Action*. 2003, Greenwich. 630.

27. MANN, K.D., *Java Server Faces in Action*. 2005, Greenwich. 1038.

28. WALLS, C., *Spring in Action*. 2008, Greenwick. 730.

29. CHRISTIAN BAUER, G.K., *Hibernate in Action*. 2005, Greenwich. 408.

30. Silberschatz, A., H.F. Korth, and S. Susarshan, *FUNDAMENTOS DE BASES DE DATOS*, ed. McGraw-Hill. 2006. 994.

31. Pecos, D. *PostGreSQL vs. MySQL*. [cited 2008 enero]; Available from: http://www.netpecos.org/docs/mysql_postgres/x57.html#AEN71.

32. Autores, C.d. *Servidores de base de datos usados en Software Libre*. 2006 [cited 2007 diciembre]; Available from: http://www.cidsol.org/downloads/arti-des-01.pdf.

33. González, C.D. *Base de Datos PostgreSQL, SQL avanzado y PHP*. 2007 [cited 2007 diciembre]; Available from: http://www.usabilidadweb.com.ar/postgre.php.

34. Franco, J.M.L. *Integración de tecnologías a través de servidores Web*. 2001 [cited 2008 enero]; Available from: http://trevinca.ei.uvigo.es/~txapi/espanol/proyecto/superior/memoria/node21.html.

35. Autores, C.d. *ARQUITECTURAS DE SOFTWARE*. 2004 [cited 2008 febrero]; Available from: http://www.cimat.mx/~trejov/Lemus2/SoftwareArchitectureAssesment.pdf.

Bibliografía

- Jacobson, Ivar; Booch, Grady; Rumbaugh, James. “El Proceso Unificado de Desarrollo de Software”. Félix Varela, 2004.
- Larman, Craig. *“Introducción al análisis y diseño orientado a objetos”*. Félix Varela. 2004. 503.
- Pressman, Roger.*” Ingeniería del Software. Un enfoque práctico”*. La Habana, Félix Varela, 2005. 601.
- Englander, Robert. *“Java and SOAP”.* O'Reilly, 2002. 276.
- Cerami, Ethan. *“Web Services Essentials”.* O'Reilly, 2002. 304.
- Colectivo de autores. *“Programación Java Server con J2EE”.* ANAYA. 1245.

Glosario de términos

Actividad Biológica: Actividad que caracteriza el comportamiento biológico en compuestos químicos.

API: Siglas en inglés de *Application Program Interface* (programa de aplicación de interfaz). Conjunto de especificaciones de comunicación entre componentes de software. Representa un método para conseguir abstracción en la programación.

Bioinformática: Es la aplicación de los ordenadores y los métodos informáticos en el análisis de datos experimentales y simulación de los sistemas biológicos.

CQF: Centro de Química Farmacéutica (CQF) es una institución para el desarrollo de investigaciones científico-tecnológicas dirigidas hacia la obtención de sustancias bio-activas para uso humano.

Compuestos Orgánicos: Compuestos cuya composición fundamental es sobre la base del elemento químico carbono.

C++: Es un lenguaje de programación, diseñado a mediados de los 80, por Bjarne Stroustrup, como extensión del lenguaje de programación C.

DAO: Siglas del inglés *Data Access Object* (Objeto de Acceso a Datos).

Descriptores: Es una forma de describir las moléculas mediante expresiones matemáticas que permite describir la estructura química o una propiedad de la molécula.

EJB: Siglas en inglés de *Enterprise JavaBeans* (*beans* empresariales de Java). Son una tecnología muy eficiente para el trabajo con sesiones, negocio, acceso a datos, invocación remota de métodos. Requieren de un tipo extra de servidor para ser ejecutados y normalmente tiende a ser muy complicado su desarrollo. Actualmente solo se utilizan en aquellos proyectos que son muy grandes.

GRID: Es una infraestructura que permite compartir recursos geográficamente dispersos para resolver problemas de gran escala. Estos recursos compartidos pueden ser hardware, software, datos e información, dispositivos electrónicos, etc.

GNU/LGPL: *"GNU Lesser General Public License"* (Licencia Pública General Menor). Pretende garantizar la libertad de compartir y modificar el software libre, esto es para asegurar que el software es libre para todos sus usuarios. Esta licencia pública general se aplica a la mayoría del software de la *"FSF Free Software Foundation"* (Fundación para el Software Libre) y a cualquier otro programa de software cuyos autores así lo establecen.

GUI: Interfaz gráfica para el usuario.

JDBC: Es el acrónimo de *Java Database Connectivity*, un API que permite la ejecución de operaciones sobre base de datos desde el lenguaje de programación Java.

Open Source: Cualidad de algunos software de incluir el código fuente en la distribución del programa. En general se usa para referirse al software libre.

Predicción: Valor estimado que caracteriza una propiedad o fenómeno obtenido de un modelo.

Plug-ins: Aplicación informática que interactúa con otra aplicación para aportarle una función o utilidad específica.

PC: Es la expresión estándar que se utiliza para denominar a las computadoras personales en general.

PC Distribuida: Computadora que se encontrará como cliente dedicado el servidor bioGRATO.

RMI: Invocación a métodos remotos, tecnología java para el trabajo distribuido.

Servlet: Objeto Java que se ejecuta dentro del contexto de un servidor web y que sirve para manipular peticiones y respuestas del cliente web. La mayor parte de los frameworks utilizan un servlet como núcleo.

SOAP: Siglas de *Simple Object Access Protocol*, protocolo simple de acceso a objetos.

Tags: Etiquetas empleadas para construir páginas web y archivos XML.

W3C: Consorcio *World Wide Web*. Es un consorcio internacional donde las organizaciones miembros, personal a tiempo completo y el público en general, trabajan conjuntamente para desarrollar estándares Web.

XML: Siglas en inglés de *eXtensible Markup Language* (lenguaje extensible de marcas).