Universidad de las Ciencias Informáticas

Facultad 6

Título: Definición de la arquitectura de un componente de comunicación de datos sobre la Web en tiempo real

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

Autor: Georvys González Rojas

Tutores: Ing. Adonis Rosales García

Ing. Jorge Bedoya Rusenko

La Habana, Junio de 2011

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

Barry Boehm, 1995

DECLARACIÓN DE AUTORÍA

Declaro ser autor del presente trabajo de diploma 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 _____.

Georvys González Rojas

_____

[Firma Autor]

Ing. Adonis Ricardo Rosales García

_____

[Firma Tutor]

Ing. Jorge Bedoya Rusenko

_____

[Firma Tutor]

DATOS DE CONTACTOS

Nombre: Georvys
Apellidos: González Rojas
Correo: grojas@estudiantes.uci.cu

Nombre: Adonis Ricardo
Apellidos: Rosales García
Título universitario: Ingeniero en Ciencias Informáticas
Año de graduado: 2008
Correo: arrosales@uci.cu

Nombre: Jorge
Apellidos: Bedoya Rusenko
Título universitario: Ingeniero en Ciencias Informáticas
Año de graduado: 2010
Correo: jbedoya@uci.cu

AGRADECIMIENTOS

A Dios por su infinito amor y porque sin él no hubiese podido ser realidad este trabajo.

A Fidel y a la Revolución por haberme permitido estudiar en esta universidad del futuro y darme la oportunidad de hacer realidad un sueño.

A mí amada madre por ser la clave de mis resultados, a Osmani por ser más que un padre y guiarme en la vida, a mi abuelita Miriam por ser mi consejera de la vida y a mi amorcito más pequeño de la vida: mi hermanita Jennifer, mi fefita.

A toda mi familia por su confianza y apoyo constante. A mi tío Eduardito, mis familias de San Miguel, de San Gregorio, de Emilio Giró, del Caribe, de Puerto Rico, mi tío Pepé, mi abuelo Nené y familiares, a mis amistades de todo el mundo.

A Maybel, mi gran amor, por su cariño y compañía, por su entrega incondicional. A su familia por ser tan especiales conmigo y haberme brindado su apoyo en todo momento. A Victoria, Delsy, Maricela, Susana, Javier niño y abuelo George, Tila y demás desde Manzanillo.

A Eduardo, Gustavo, Elio, Yeymi, Manuel, Orea, Erio, Yuniesky, Reynaldo, Marislay y todos mis amigos del ajedrez.

A mis tutores Adonis y Jorge, a Garnache y demás compañeros del proyecto.

A los profesores y amigos de aula y de esta Universidad.

Al tribunal este trabajo y su oponente.

A mis amigos y vecinos de Guantánamo.

A todas las personas que de una forma u otra hicieron posible este resultado.

DEDICATORIA

Este trabajo está especialmente dedicado a mi madre Leyanis Rojas Olivares por ser lo que más quiero en la vida.

A mi papá Osmani por sacrificarse junto a mi madre para ser realidad este sueño.

A mi hermanita Jennifer y mi abuelita Miriam por estar siempre presente en mi corazón.

RESUMEN

A partir de la necesidad que surge en el Centro de Tecnologías de Gestión de Datos (DATEC) de la facultad 6 de lograr la comunicación y visualización gráfica de indicadores claves en tiempo real, nace el presente trabajo.

La arquitectura diseñada facilita y agiliza la implementación de una aplicación para el proceso de toma de decisiones en cualquier organización. Precisamente el objetivo de este trabajo radica en diseñar la arquitectura de un componente de comunicación de datos sobre la web en tiempo real para un *Dashboard*.

Para la selección de las herramientas informáticas y tecnologías para el desarrollo del sistema se tuvo en cuenta la situación del país en materia de acceso a las herramientas propietarias y se decidió seguir la línea de aquellas que estuvieran bajo licencia libre.

PALABRAS CLAVES

Arquitectura, toma de decisiones, componente de comunicación, tiempo real, herramientas y tecnologías, libre.

ÍNDICE

INTRODUCCIÓN

Las organizaciones se desarrollan en la actualidad en un entorno de mayor competencia, por lo que se hace necesario no solo permanecer, sino también crecer e innovar constantemente para ser más eficiente y competitivos en el mercado, de tal manera que contribuya al éxito de la empresa en el corto, mediano y largo plazo. Las técnicas y tecnologías utilizadas para ello han tenido que evolucionar progresivamente. Se han favorecido de internet para el intercambio de datos en el mundo y de nuevas herramientas de apoyo a los procesos, como en el caso de los sistemas de control de gestión.

Ante la necesidad de completar la perspectiva financiera tradicional de medición del éxito de las organizaciones, en 1992 surge una nueva herramienta, el Cuadro de Mando Integral (CMI) o *Balanced Scorecard* (BSC) como también es conocido en su versión en inglés. CMI es un recurso fundamental en el actual ambiente de rápidos cambios tecnológicos, y en el que se ha hecho prioritario a la organización medir los resultados financieros, satisfacción del cliente, operaciones y la capacidad de la organización para producir y ser competitiva. Sus autores, Robert Kaplan y David Norton, plantean que el CMI es un sistema de administración o sistema administrativo (*Management system*), que va más allá de la perspectiva financiera con la que los gerentes acostumbran evaluar la marcha de una empresa [1]. Estos autores proponen una serie de indicadores genéricos que cada organización debe adaptar a sus propias necesidades, ellas son, perspectiva financiera, perspectiva del cliente, perspectiva interna o de procesos de negocio y perspectiva de desarrollo y aprendizaje [1].

En la actualidad no todos los Cuadros de Mando Integral están basados en los principios de Kaplan y Norton, aunque sí influenciados en alguna medida por ellos. Por este motivo, se suele emplear con cierta frecuencia el término *Dashboard*, que refleja algunas características teóricas del Cuadro de Mando. De forma genérica, un *Dashboard* [2] engloba varias herramientas que muestran información relevante para la empresa a través de una serie de indicadores de rendimiento, también denominados KPIs (*Key Performance Indicators*). Así esta información que proviene de diversas fuentes: *ERPs*, *Data Warehouses*, sistemas de producción o cualquier otra fuente de datos, se presenta a través de componentes gráficos que permiten visualizar rápidamente el estado de los procesos clave para la organización.

Cuba no se encuentra ajena a los avances tecnológicos producidos en el mercado mundial del software. Por esta razón se ha aplicado el uso de los CMI paulatinamente en algunas instituciones, en

las que una vez creado, se han observado resultados satisfactorios para la economía cubana, ejemplo de ello lo constituye CUBACEL, ASTICAR, GET Varadero, SEPSA Cienfuegos, Rado y Asociados, Intermar Cienfuegos, ETECSA SA, CUPET, ESEN, CENEX, ESIC, Hotel Plaza, EMPAI de Matanzas. [3]

La Universidad de las Ciencias Informáticas (UCI), 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 la Oficina Nacional de Estadísticas (ONE) en la creación de varios proyectos informáticos. En la facultad seis el Centro de Tecnologías de Gestión de Datos (DATEC) participa directamente en el desarrollo de diversos productos para el uso nacional e internacional. Algunos de ellos necesitan de una herramienta libre que sea capaz de medir a modo de gráficos información de la empresa extraída de varias fuentes o bases de datos. Por lo que se hace necesaria una tecnología web que soporte la comunicación en tiempo real, donde las vistas gráficas se actualicen inmediatamente después que se actualicen los datos. Ante la necesidad planteada, el centro DATEC actualmente no cuenta con una arquitectura definida como guía para llevar a cabo la implementación de dicha tecnología.

Teniendo en cuenta lo anteriormente analizado se plantea como problema científico de la investigación: ¿Cómo lograr la comunicación y visualización gráfica de indicadores claves en tiempo real sobre la web para un *Dashboard*?

La investigación tiene como objeto de estudio la Arquitectura de Software (AS), cuyo campo de acción está enmarcado en La arquitectura de software para componente de comunicación de datos sobre la web en tiempo real para un *Dashboard*.

Para dar solución al problema científico se propone como objetivo general: Diseñar la arquitectura de un componente de comunicación de datos sobre la web en tiempo real. La consecución de tal objetivo estará sustentada en los objetivos específicos siguientes:

1. Seleccionar estilos y patrones de arquitectura.

2. Describir la arquitectura del sistema.

3. 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. Identificación de los estilos arquitectónicos a utilizar, fundamentación de estos estilos, así como su aplicación.
2. Identificación de los patrones arquitectónicos a utilizar, fundamentación de estos patrones, así como su aplicación.
3. Definición y fundamentación de las herramientas a utilizar para el desarrollo de la plataforma.
4. Identificación de los protocolos de comunicación adecuados.
5. Análisis y diseño de las vistas del sistema.
6. Representación del diseño arquitectónico propuesto.
7. Selección y aplicación de un método para validar el diseño arquitectónico propuesto.

El trabajo de diploma está estructurada de la siguiente manera: resumen, introducción, tres capítulos, conclusiones, recomendaciones, glosario de términos, referencias bibliográficas y bibliografía.

Capítulo 1: Fundamentación Teórica de la investigación. En este capítulo se brinda una descripción general del objeto de estudio (AS) y los sistemas existentes asociadas al campo de acción. Se plantea un análisis exhaustivo de varios conceptos asociados al objeto de estudio, se definen las técnicas, tecnologías y metodologías que se utilizan para dar solución al problema planteado.

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.

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 1: FUNDAMENTACIÓN TEÓRICA DE LA INVESTIGACIÓN

Introducción

En este capítulo se brinda una descripción general del objeto de estudio (AS) y los sistemas existentes asociadas al campo de acción. Se plantea un análisis exhaustivo de varios conceptos asociados al objeto de estudio, se definen las técnicas, tecnologías y metodologías que se utilizan para dar solución al problema planteado.

1.1 Definición de arquitectura de software

Existen muchas definiciones de arquitectura de software como producto de la discrepancia de los arquitectos. Entre las definiciones que se consideran las más aceptadas se encuentran:

Según Philippe Kruchten:

"La arquitectura de software, tiene que ver con el diseño y la implementación de estructuras de software de alto nivel. Es el resultado de ensamblar un cierto número de elementos arquitectónicos de forma adecuada para satisfacer la mayor funcionalidad y requerimientos de desempeño de un sistema, así como requerimientos no funcionales, como la confiabilidad, escalabilidad, portabilidad, y disponibilidad".[4]

Len Bass, Paul Clements y Rick Kazman plantean:

"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." [5]

Para la investigación se coincide con la propuesta oficial del documento Std 1471-2000 del *Institute of Electrical and Electronics Engineers* (lEEE), 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". [6]

La arquitectura de software se ve influenciada por muchos factores, como la plataforma funcional del software (arquitectura de hardware, sistema operativo, sistemas de gestión de base de datos, protocolos para comunicaciones en red), los bloques de construcción reutilizables de que se dispone, un marco de trabajo para interfaces gráficas de usuario, consideraciones de implantación, sistemas heredados, y requisitos no funcionales (rendimiento y fiabilidad). [7]

El tópico más urgente y exitoso en arquitectura de software en los últimos cuatro o cinco años es, sin duda, el de los patrones, tanto en lo que concierne a los patrones de diseño como a los de arquitectura. Inmediatamente después, en una relación a veces de complementariedad, otras de oposición, se encuentra la sistematización de los llamados estilos arquitectónicos [8]. A continuación se describen las características principales del sistema para así dar paso a las cuestiones arquitectónicas que generan como son, los estilos y patrones arquitectónicos.

1.2 Características del sistema

El sistema que se desea construir, constituirá una potente herramienta capaz de medir a modo de gráficos información actualizada de los procesos que se desarrollan en la empresa, y una acertada base sobre la cual tomar decisiones.

El sistema debe permitir de manera integrada, la realización de diferentes operaciones, entre las que identifican más con un sistema de tiempo real se encuentran: conectarse a un servidor XMPP (servicio PubSub), obtener nodos, suscribirse a un nodo y publicar eventos a un servicio de tal manera que todos los abonados a un nodo puedan recibir la notificación del evento. Todas estas acciones entre otras, deben estar estrechamente vinculadas en un mismo entorno para de esta forma permitir un mejor desempeño del usuario con la aplicación.

1.3 Estilos arquitectónicos

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]

Existen muchas otras definiciones de estilos, cabe citar que en 1996 Robert Allen y David Garlan [10] asimilan los estilos arquitectónicos a descripciones informales de arquitectura basadas en una colección de componentes computacionales, junto a una colección de conectores que describen las interacciones entre los componentes. Consideran que esta es una descripción deliberadamente abstracta, que ignora aspectos importantes de una estructura arquitectónica, tales como una descomposición jerárquica, la asignación de computación a los procesadores, la coordinación global y el plan de tareas.

En 1999 Mark Klein y Rick Kazman proponen una definición según la cual un estilo arquitectónico es una descripción del patrón de los datos y la interacción de control entre los componentes, ligada a una descripción informal de los beneficios e inconvenientes aparejados por el uso del estilo. Los estilos arquitectónicos, afirman, son artefactos de ingeniería importantes porque definen clases de diseño junto con las propiedades conocidas asociadas a ellos. Ofrecen evidencia basada en la experiencia sobre la forma en que se ha utilizado históricamente cada clase, junto con razonamiento cualitativo para explicar por qué cada clase tiene esas propiedades específicas. [11]

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 que se exponen a continuación:

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.

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.

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.

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

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.

Arquitecturas en Capas

El estilo arquitectural en capas se basa en una distribución jerárquica de los roles y las responsabilidades para proporcionar una división efectiva de los problemas a resolver. Los roles indican el tipo y la forma de la interacción con otras capas y las responsabilidades la funcionalidad que implementan. [12]

Ventajas

Soporta un diseño basado en niveles de abstracción crecientes, lo cual a su vez permite a los implementadores la partición de un problema complejo en una secuencia de pasos incrementales. Admite muy naturalmente optimizaciones y refinamientos. 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 de este estilo son los objetos, o más bien instancias de los tipos de dato abstractos. Los objetos representan una clase de componentes llamados managers, debido a que son responsables de preservar la integridad de su propia representación [9, pág. 22]. Un rasgo importante de este aspecto es que la representación interna de un objeto no es accesible desde otros objetos.

Ventajas

Entre las cualidades la más básica concierne a que se puede modificar la implementación de un objeto sin afectar a sus clientes. De este modo es posible descomponer problemas en colecciones de

agentes en interacción.

Desventajas

Entre las limitaciones, el principal problema del estilo se manifiesta en el hecho de que para poder interactuar con otro objeto a través de una invocación de procedimiento, se debe conocer su identidad. Los componentes se encuentran fuertemente acoplados y sólo soportan interacciones atómicas.

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. En términos de patrones de diseño, el patrón que corresponde más estrechamente a este estilo es el que se conoce como *Observer*. La idea dominante en la invocación implícita, como se le ha llamado también a las arquitecturas basadas en eventos, 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.*
- Puede mejorar eficiencia, eliminando la necesidad de *polling* por ocurrencia de evento
- Es fácil de empaquetar en una transacción atómica.
- Se puede agregar un componente registrándolo para los eventos del sistema; se pueden reemplazar componentes.
- Modularidad: una sola modalidad para eventos diversos.
- Es agnóstica en lo que respecta a si las implementaciones corren sincrónica o asincrónicamente porque no se espera una respuesta.

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.
- Posibilidad de desborde.
- Potencial imprevisión de escalabilidad.
- No hay mucho soporte de recuperación en caso de falla parcial.

Luego de un análisis realizado a partir de los estilos anteriormente descritos y las características del sistema, de complejidad y alto nivel de diseño se determinó que la aplicación requiere la separación en partes que puedan concentrarse en distintas áreas de funcionalidad. Por lo anteriormente mencionado se propone aplicar dentro de la familia de estilos de llamada y retorno, el estilo de arquitecturas en capas. Es importante que una arquitectura no se base en un solo estilo [12] arquitectural, sino que combine varios de dichos estilos para obtener las ventajas de cada uno. Para esta investigación se realiza la combinación del estilo de arquitectura en capas con el estilo de arquitecturas basadas en eventos que pertenece a la familia de estilos *Peer to Peer*. Se hace necesario aplicar este último ya que procura mejorar la eficiencia, eliminando la necesidad de *polling* por ocurrencia de evento, así como otras ventajas que brinda dicho estilo.

1.4 Patrones

Las definiciones relacionadas con los patrones y las prácticas son diversas, y luego de un estudio minucioso de los autores del tema no se ha podido encontrar dos caracterizaciones que reconozcan las mismas clases o que se sirva de una terminología estable [9, pág. 41].

Una de las bibliografías más enriquecidas en el tema de los patrones es el texto *Pattern-oriented software architecture* (POSA) de Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad y Michael Stale. En él los patrones arquitectónicos son aproximadamente lo mismo que lo que se acostumbra definir como estilos y ambos términos se usan de manera indistinta. En POSA los patrones expresan un esquema de organización estructural para los sistemas de software.

Proporcionan un conjunto de subsistemas predefinidos, especifica sus responsabilidades e incluye reglas y lineamientos para organizar la relación entre ellos.

Los patrones son formas de describir las mejores prácticas, buenos diseños, y encapsulan la experiencia de forma tal que es posible para otros el reutilizar dicha experiencia. Constituyen mecanismos cuyo objetivo es la solución de problemas que ocurren repetidamente dentro de un contexto muy bien definido. Las soluciones que son propuestas a través de patrones involucran algunas clases de estructuras que permiten contemplar los requisitos no funcionales.

Entre los principales patrones se encuentran: [13]

Patrones de arquitectura, relacionados a la interacción de objetos dentro o entre los niveles arquitectónicos, se aplican durante la fase de diseño inicial para resolver problemas arquitectónicos, adaptabilidad a requerimientos cambiantes, performance, modularidad y acoplamiento. Constituyen patrones de llamada entre objetos, similar a los patrones de diseño, decisiones y criterios arquitectónicos, así como el empaquetado de funcionalidad sobresalen como sus principales soluciones.

Patrones de diseño, se aplican durante la fase de diseño detallado para solucionar problemas de claridad del diseño, multiplicación de clases, adaptabilidad a requerimientos cambiantes.

Patrones de análisis, usualmente específicos de aplicación, se aplican durante el análisis para solucionar problemas relacionados con el modelo de dominio, completitud, integración y equilibrio de objetivos múltiples, planeamiento para capacidades adicionales comunes.

Patrones de proceso o de organización, desarrollo o procesos de administración de proyectos, técnicas o estructura de organización, se aplican durante la fase de planeamiento para solucionar problemas con la productividad, comunicación efectiva y eficiente.

Patrones de idiomas, se aplican durante las fases de desarrollo de implementación, despliegue y mantenimiento, constituyen estándares de codificación y proyecto, sumamente específicos de un lenguaje, plataforma o ambiente y están encaminados a solucionar los problemas con la legibilidad, predictibilidad y operaciones comunes bien conocidas en un nuevo ambiente. Regulan la nomenclatura en la cual se escriben, se diseñan y desarrollan los sistemas.

Luego de comparar los principales patrones, se determinó que los patrones de arquitectura y de

diseño son parte de los mecanismos arquitectónicos a seguir para el desarrollo de esta investigación.

1.5.1 Patrones de Arquitectura

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.[14]

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.

1.5.1.1 Patrón MVC

El patrón Modelo-Vista-Controlador (MVC) es muy recomendado para sistemas interactivos. Actualmente los sistemas informáticos siguen los patrones arquitectónicos en tres capas, el orientado a objetos y el modelo-vista-controlador. Fue un concepto introducido por los creadores del lenguaje *Smalltalk* en los años 80, con la idea de agrupar las clases en función del rol que desempeñan en la aplicación. Es un patrón de arquitectura de software que separa los datos de una aplicación, la interfaz de usuario, y la lógica de control en tres componentes distintos. Está formado por tres niveles:

- El modelo representa la información con la que trabaja la aplicación, es decir, su lógica de negocio.
- La vista transforma el modelo en una página web que permite al usuario interactuar con ella.
- El controlador se encarga de procesar las interacciones del usuario y realiza los cambios apropiados en el modelo o en la vista.

La arquitectura MVC separa la lógica de negocio (el modelo) y la presentación (la vista) por lo que se consigue un mantenimiento más sencillo de las aplicaciones. El controlador se encarga de aislar al modelo y a la vista de los detalles del protocolo utilizado para las peticiones (HTTP, consola de comandos y email). El modelo se encarga de la abstracción de la lógica relacionada con los datos, haciendo que la vista y las acciones sean independientes de, por ejemplo, el tipo de gestor de bases de datos utilizado por la aplicación. [15]

1.5.1.2 Patrón de Arquitectura en Capas

Los autores Garlan y Shaw [16] definen el estilo en capas 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.

En algunos ejemplares, las capas internas están ocultas a todas las demás, menos para las capas externas adyacentes, y excepto para funciones puntuales de exportación; en estos sistemas, los componentes implementan máquinas virtuales en alguna de las capas de la jerarquía. En otros sistemas, las capas pueden ser sólo parcialmente opacas. En la práctica, las capas suelen ser entidades complejas, compuestas de varios paquetes o subsistemas.

Entre los principales beneficios de una arquitectura en capas se encuentran: [12]

- Abstracción ya que los cambios se realizan a alto nivel y se puede incrementar o reducir el nivel de abstracción que se usa en cada capa del modelo.
- Aislamiento ya que se pueden realizar actualizaciones en el interior de las capas sin que esto afecte al resto del sistema.
- Rendimiento ya que distribuyendo las capas en distintos niveles físicos se puede mejorar la escalabilidad, la tolerancia a fallos y el rendimiento.
- Testeabilidad ya que cada capa tiene una interfaz bien definida sobre las que realizar las pruebas y la habilidad de cambiar entre diferentes implementaciones de una capa.
- Independencia ya que elimina la necesidad de considerar el hardware y el despliegue así como las dependencias con interfaces externas.

Las capas son agrupaciones horizontales lógicas de componentes de software que forman la aplicación o el servicio. Permite diferenciar entre los diferentes tipos de tareas a ser realizadas por los componentes, ofreciendo un diseño que maximiza la reutilización y especialmente la mantenibilidad.

En definitiva se trata de aplicar el principio de separación de responsabilidades dentro de una arquitectura.

Cada capa lógica de primer nivel puede tener un número concreto de componentes agrupados en sub-capas. Dichas sub-capas realizan a su vez un tipo específico de tareas. Cada capa de primer nivel debe de estar débilmente acoplada con el resto de las capas de primer nivel.

En soluciones grandes que involucran a muchos componentes de software, es habitual tener un gran número de componentes en el mismo nivel de abstracción (capas) pero que sin embargo no son cohesivos. En esos casos cada capa debería descomponerse en dos o más subsistemas cohesivos.

La descomposición más habitual es dividir la aplicación en tres capas:

1. Lógica de presentación, que se ocupa de toda la interacción entre el usuario y el software, pudiendo tratarse de un sistema de menús muy simple o una interfaz gráfica de usuario relativamente compleja.
2. La lógica del dominio, también conocida como la lógica de negocio, que es todo lo que necesita conocer la aplicación para poder trabajar con el dominio en cuestión. Implica realizar cálculos basados en datos de entrada o almacenados, validación de cualquier dato proveniente de la capa de presentación y la ejecución de algoritmos específicos en función de los comandos de la presentación.
3. La lógica de fuente de datos, que se ocupa de comunicarse con otros sistemas que se encargan de tareas en representación de la aplicación. Estos pueden ser monitores de transacciones, otras aplicaciones y sistemas de mensajes.

La descomposición en tres capas de la aplicación es precisamente la adoptada por la investigación para su diseño arquitectónico ya que se permite encapsular aplicaciones complejas. Como se trata de un componente de comunicación en tiempo real es factible emplear dicha arquitectura para el software que se pretende diseñar. Esta permite mejorar el mantenimiento del código, permite diferentes tipos de despliegue proporcionando una clara delimitación de donde debe estar cada tipo de componente funcional e incluso cada tipo de tecnología. Las capas deben estar débilmente acopladas entre ellas y con alta cohesión internamente.

1.5.2 Patrones de Diseño

Un patrón de diseño constituye un esquema para refinar subsistemas o componentes. Es una descripción de clases y objetos comunicándose entre sí adaptada para resolver un problema de diseño general en un contexto particular. Un patrón de diseño identifica: clases, instancias, roles, colaboraciones y la distribución de responsabilidades, además de que ayuda a construir clases y a estructurar sistemas de clases. Los patrones de diseño son la base para la búsqueda de soluciones a problemas comunes en el desarrollo de software y otros ámbitos referentes al diseño de interacción o interfaces.

1.5.2.1 Patrones GRASP

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 cinco patrones [17] principales, que son: el patrón Experto, Creador, Alta cohesión, Bajo acoplamiento y Controlador. A continuación se exponen las características de cada uno de ellos.

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. [17, pág. 196]

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).

El patrón Creador [17, pág. 198] 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.

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.

El Bajo Acoplamiento [17, pág. 201] es un principio que no se puede 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.

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).

Como el patrón Bajo Acoplamiento, también Alta Cohesión [17, pág. 204] 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.

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.

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.

La mayor parte de los sistemas reciben eventos de entrada externa [17, pág. 208], 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.

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.

1.5.2.2 Patrones GOF

El catálogo de patrones más famoso es el contenido en el libro *"Design Patterns: Elements of Reusable Object-Oriented Software"*, el de la banda de los 4, también conocido como el libro GOF *(Gang-Of-Four Book).* Según el libro GOF [18] estos patrones se clasifican según su propósito en creacionales, estructurales y de composición, mientras que respecto a su ámbito se clasifican en clases y objetos:

Respecto a su propósito:

1. **De Creación:** Abstraen el proceso de creación de instancias. Resuelven problemas relativos a la creación de objetos.
2. **Estructurales:** Se ocupan de cómo clases y objetos son utilizados para componer estructuras de mayor tamaño. Resuelven problemas relativos a la composición de objetos.
3. **De Comportamiento:** Atañen a los algoritmos y a la asignación de responsabilidades entre objetos. Resuelven problemas relativos a la interacción entre objetos.

En el Anexo 1 se muestra una breve descripción de los patrones GOF respecto a su propósito.

Respecto a su ámbito:

1. **Clases:** Relaciones estáticas entre clases
2. **Objetos:** Relaciones dinámicas entre objetos.

Luego de realizar un estudio de los patrones se determinó como patrón de arquitectura el patrón en capas como esquema de organización estructural para el sistema de software. Dentro de los patrones de diseños analizados se propusieron los patrones GRASP ya que permiten diseñar eficazmente el software. Estos patrones describen los principios fundamentales de la asignación de responsabilidades a objetos en forma de patrones. La familia de patrones GOF es muy amplia y se propusieron en esta investigación los patrones *Decorator* y *Observer*. Con el objetivo de agregar responsabilidades o comportamiento de manera dinámica a un objeto en específico se definió utilizar el patrón *Decorator*.

El *observer* define una relación de un objeto a muchos objetos, de manera que cuando uno de los objetos cambia su estado, el observador se encarga de notificar este cambio a todos los otros objetos. Este patrón es fundamental para el componente que se pretende diseñar por las características presentadas anteriormente en este mismo documento. Dicho patrón suele observarse en los frameworks de interfaces gráficas [12, pág. 335] orientados a objetos en los que la forma de capturar eventos es suscribir a los objetos que pueden disparar eventos.

El patrón *observer* suele relacionarse con la publicación-suscripción por la forma en que se manipula la información de los objetos, así como sus relaciones. A continuación se presenta una descripción del Paradigma Publicación-Suscripción.

1.6 Paradigma Publicación-Suscripción

El modelo de Publicación-Suscripción (PubSub) se trata de un paradigma de envió de mensajes asíncrono mediante el cual los usuarios que publican información (productor) no la envían directamente a los usuarios que solicitan esta información (consumidor), sino que lo hacen mediante un mediador o *broker*, ver figura 1. [19]

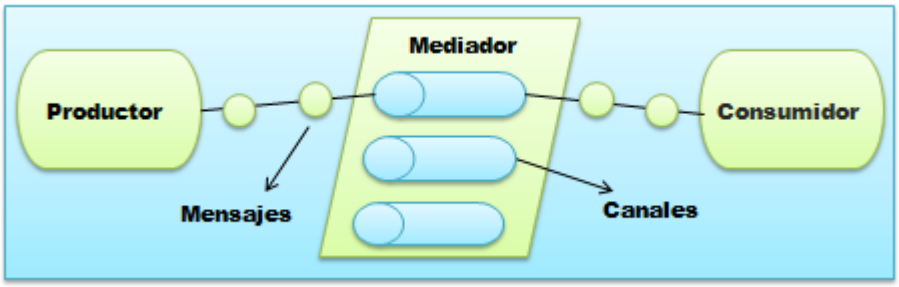

Figura 1 Paradigma de Publicación-Suscripción

Productor de información: Usuario que tiene la información a difundir. El productor publica la información en el mediador sin tener conocimiento alguno de los usuarios interesados en esa información.

Consumidor de la información: Usuario interesado en recibir información. El consumidor se suscribe a los temas en los que está interesado y cuando el productor publica información, el consumidor recibe una notificación indicando que hay nueva información disponible.

Mediador (o broker): Está situado entre el productor y el consumidor, y se encarga de recibir la información de los productores y las peticiones de suscripción de los consumidores. Además, también se encarga de enviar a los consumidores las notificaciones precisas cuando un productor publica nuevos contenidos.

Canal: Se trata de los conectores lógicos entre el productor y el consumidor. El canal determina algunas de las propiedades de la información (tipo de información, formato entre otras). Además también gestiona la forma como se distribuyen los contenidos, si la información expira o es persistente, o si los datos se entregan en el momento de la publicación, o por el contrario el consumidor puede solicitarlos cuando quiera, independientemente del momento en que se generaron.

Este sistema envía la información sólo a los que están interesados en recibirla, evitando así búsquedas innecesarias en la red. Esta disociación de los productores y los consumidores pueden permitir una mayor escalabilidad y una mayor dinámica topológica de la red. Este paradigma es de vital importancia para el componente de comunicación en tiempo real y es por ello que se decide utilizar tecnologías que lo implementen para lograr el objetivo, más adelante en este mismo capítulo quedan planteadas las tecnologías a utilizar.

1.7 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 por Grady Booch, Ivar Jacobson y Jim Rumbaugh, estos autores fueron contratados por la empresa *Rational Software Company*. 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. [20]

Es la especificación de OMG (Grupo de gestión de objetos, en inglés *Object Management Group*) y no solo es la forma mundial de representar la estructura de las aplicaciones, comportamiento y arquitectura sino que también representa procesos de negocio y estructura de datos. [21]

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. Este lenguaje tiene elementos que conforman su vocabulario. En los diagramas se relacionan estos conjuntos de elementos y son los encargados de visualizar un sistema desde diferentes perspectivas.

¿Por qué 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 las personas involucradas en el desarrollo de un sistema (ingenieros, científicos del área de computación, administradores, líderes y otros), o para la documentación de software existente. Por tales razones antes mencionadas se seleccionó UML 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 lenguaje de descripción de la arquitectura.

1.8 Metodología de desarrollo de software

La metodología de software se desarrolla con el objetivo de dar solución a los problemas existentes en la producción de software, que cada vez son más complejos. Estas engloban procedimientos,

técnicas, documentación y herramientas que se utilizan en la creación de un producto de software [22]. Una metodología es un proceso. Existen estándares y procesos de desarrollo de software que determinan buenas prácticas para el posterior desarrollo de las aplicaciones. El siguiente estudio fue realizado con las metodologías más utilizadas.

1.8.1 Proceso Unificado de Desarrollo (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 integrador y propone un modelo de referencia organizacional del personal. Utiliza UML como fundamental lenguaje de modelado para el desarrollo de todos los modelos. [7]

En RUP se han agrupado las actividades en grupos lógicos definiéndose 9 flujos de trabajo principales. Los 6 primeros son conocidos como flujos de trabajo de ingeniería y los tres últimos como flujos de trabajo de apoyo.

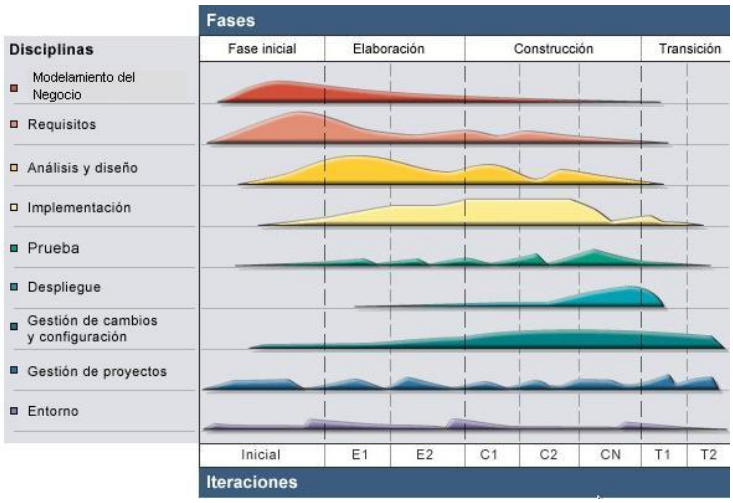

Figura 2: RUP en dos dimensiones. Se representa el proceso en el que se grafican los flujos de trabajo y las fases y muestra la dinámica expresada en iteraciones y puntos de control.

El ciclo de vida de RUP se caracteriza por ser:

Dirigido por casos de uso: Los casos de uso reflejan lo que los usuarios futuros necesitan y desean, lo cual se capta cuando se modela el negocio y se representa a través de los requerimientos. A partir de aquí los casos de uso guían el proceso de desarrollo ya que los modelos que se obtienen, como

resultado de los diferentes flujos de trabajo, representan la realización de los casos de uso (cómo se llevan a cabo).

Centrado en la arquitectura: La arquitectura muestra la visión común del sistema completo en la que el equipo de proyecto y los usuarios deben estar de acuerdo, por lo que describe los elementos del modelo que son más importantes para su construcción, los cimientos del sistema que son necesarios como base para comprenderlo, desarrollarlo y producirlo económicamente. RUP se desarrolla mediante iteraciones, comenzando por los CU relevantes desde el punto de vista de la arquitectura.

Iterativo e Incremental: RUP propone que cada fase se desarrolle en iteraciones. Una iteración involucra actividades de todos los flujos de trabajo, aunque desarrolla fundamentalmente algunos más que otros. Por ejemplo, una iteración de elaboración centra su atención en el análisis y diseño, aunque refina los requerimientos y obtiene un producto con un determinado nivel, pero que irá creciendo incrementalmente en cada iteración.

1.8.2 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: [23]

1. 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.

2. 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.
3. 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.

4. Evolución continúa 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 cómo 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 el desarrollo de los proyectos en el centro DATEC se determinó utilizar esta metodología para este trabajo.

1.9 Rol arquitecto de software

El rol arquitecto de software dentro de un proyecto de desarrollo de software tiene las siguientes responsabilidades: liderar y coordinar actividades técnicas, liderar el proceso de arquitectura y

producir los artefactos necesarios durante el ciclo de vida del proyecto: Línea base de la arquitectura, Documento de Descripción de Arquitectura de Software, Informe del Levantamiento de Información para la Arquitectura de Información, Arquitectura de Información, modelo de análisis, modelo de diseño, modelo de despliegue, modelo de implementación, prototipos de arquitectura, visualizar el comportamiento del sistema, crear los planos del sistema, definir la forma en la cual los elementos del sistema trabajan en conjunto, proveer una guía técnica clara y consistente. Establece la estructura global de cada vista de la arquitectura: la descomposición de las vistas, el agrupamiento de elementos y las interfaces de los mismos.

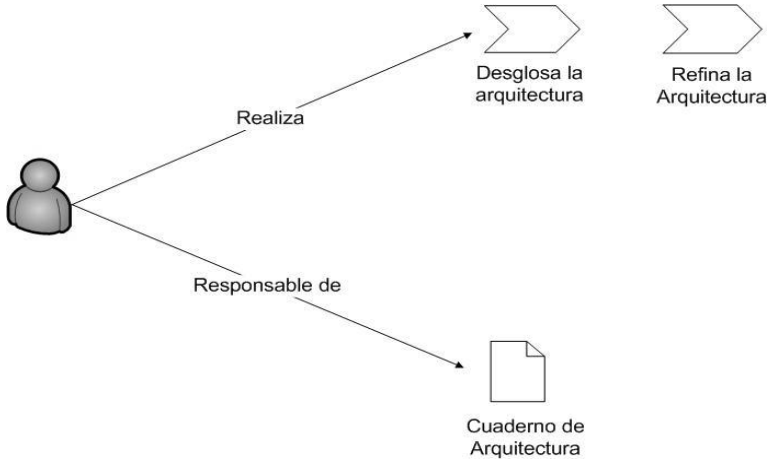

Figura 3: Funciones del arquitecto según *OpenUP*

Según la metodología *OpenUP* entre las funciones del arquitecto de software se encuentran realizar las actividades de desglose de la arquitectura y refinamiento de la misma; 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 Lenguajes, tecnologías y herramientas de soporte al desarrollo

1.10.1 Herramientas Case para el desarrollo de software

Las herramientas CASE (*Computer Aided Software Engineering*, Ingeniería de Software Asistida por Ordenador) son diversas aplicaciones informáticas destinadas a aumentar la productividad en el desarrollo de software reduciendo el coste de las mismas en términos de tiempo y de dinero. Estas herramientas pueden ayudar en todos los aspectos del ciclo de vida de desarrollo del software en tareas como el proceso de realizar un diseño del proyecto, cálculo de costes, implementación de parte del código automáticamente con el diseño dado, compilación automática y documentación o detección de errores.

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. [24]

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 6.1

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.

Existen otras herramientas CASE a parte de las antes mencionadas pero la tomada en cuenta para el trabajo arquitectónico 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.

1.10.2 Lenguaje de programación

Uno de los aspectos importantes a tener en cuenta en un lenguaje de programación es la portabilidad. Poder usar la misma aplicación en distintas arquitecturas o sistemas operativos sin tener que recompilar supone un gran ahorro de desarrollo. A continuación se presentan las características más importantes de algunos lenguajes de programación y el propuesto para la investigación.

Java

Ofrece toda la funcionalidad de un lenguaje potente. 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.

En su curso de java Salazar [25] explica detalladamente otras cuestiones del lenguaje como robustez, multiplataforma, seguridad, multitarea, lenguaje orientado a objeto y simple de aprender.

Phyton

Es uno de los lenguajes de script más frecuentemente empleados, se destaca por estar orientado a objetos de principio a fin. Su sintaxis emplea tabuladores para marcar bloques de código, destaca por la claridad y legibilidad de sus programas. Contiene una estructura mínima, ya que todo el lenguaje está desarrollado a partir de componentes básicos, los cuales también pueden ser modificados.

PHP5

PHP es un acrónimo recursivo que significa PHP *Hypertext Pre-processores*. Es un lenguaje interpretado de alto nivel embebido en páginas HTML y ejecutado en lado del servidor. Puede ser desplegado en la mayoría de los servidores web y en casi todos los sistemas operativos y plataformas sin costo alguno. Este lenguaje posee amplias ventajas como:

- Es un lenguaje multiplataforma.
- Capacidad de conexión con la mayoría de los manejadores de base de datos que se utilizan en la actualidad, destaca su conectividad con MySQL.
- Capacidad de expandir su potencial utilizando la enorme cantidad de módulos (llamados ext's o extensiones).
- Posee una amplia documentación en su página oficial (www.php.net), entre la cual se destaca que todas las funciones del sistema están explicadas y ejemplificadas en un único archivo de ayuda.
- Es libre, por lo que se presenta como una alternativa de fácil acceso para todos.
- Permite las técnicas de Programación Orientada a Objetos.

- Contiene una biblioteca nativa de funciones sumamente amplia e incluida.
- No requiere definición de tipos de variables.
- Tiene manejo de excepciones (desde PHP5). [26]

PHP es un lenguaje sencillo, de sintaxis cómoda y dispone de muchas librerías que facilitan en gran medida el desarrollo de las aplicaciones; como es el caso de un componente de tiempo real. El lenguaje en cuestión es el que se propone por esta investigación para llevar a cabo su solución.

1.10.3 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. [27]

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:

- 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:

- Mantiene versiones no solo 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 ya que esta herramienta reduce las inconsistencias y agiliza las transferencias y actualizaciones de datos. Este sistema 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.

1.10.4 Frameworks JavaScript

En la actualidad existen muchos frameworks en JavaScript y es muy difícil determinar cuál es mejor. Es importante recordar que estos frameworks son simples herramientas que ayudan a realizar diferentes tareas de diferentes tipos, todos basados en el mismo lenguaje, javascript. Lo correcto es seleccionar uno u otro dependiendo la utilidad y la capacidad de saber cómo utilizarlo.

Entre los populares Frameworks para Javascript se tienen jQuery, Prototype, MooTools, ExtJS, Dojo y otros muchos más que se encuentran disponibles en múltiples sitios web del internet. Para la

investigación se tuvieron en cuenta ExtJS y Dojo. En lo adelante se muestran algunas de las características de estos dos frameworks.

ExtJS 3.0

ExtJS es un framework de presentación completamente OO (Orientado a Objetos) de JavaScript, es multiplataforma y hace uso de la tecnología AJAX. Este framework brinda múltiples posibilidades para el trabajo con las validaciones y manejo de errores en el cliente. Además permite la personalización de temas de estilos y provee el trabajo con una amplia configuración e intenso trabajo con las hojas de estilo CSS. Basa toda su funcionalidad en JavaScript a través de librerías YUI, jQuery, o haciendo uso de la librería nativa, así en tiempo de ejecución carga y crea todos los objetos HTML a través del uso intenso de DOM. Cuenta con dos licencias, una comercial y otra Open Source. [28]

Dojo 1.3

Dojo es otro Framework en JavaScript el cual permite añadir características dinámicas fácilmente a las páginas Web. Puedes utilizar los componentes que incorpora Dojo para mejorar la usabilidad y funcionabilidad. Se puede construir interfaces con degradados de color, widges y transacciones animadas de forma rápida y sencilla. Contiene varias características que trabajan a través de la mayoría de los navegadores, tales como: Menúes, Tabs, Tooltips y Tablas ordenables.

La versión 3.0 de Extjs cuenta con una cantidad muy limitada de gráficos y pocas posibilidades de animación. Se determinó utilizar Dojo para la presentación de interfaz de usuario y la visualización de los gráficos que actualicen los datos en tiempo real.

1.10.5 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 6.7.1

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 NetBeans IDE, es un producto libre y gratuito sin restricciones de uso [29]. Todas las funciones del IDE son provistas por módulos. Cada módulo provee una función bien definida.

NetBeans permite crear aplicaciones con python ya que posee un motor para escribir (resaltando la sintaxis), identificar errores y el debugger. Sin duda alguna, netbeans se ha convertido en un IDE apto para la mayoría de los lenguajes de programación de código abierto. También se estima que dará para soporte GUI para varias librerías gráficas como son PyQt y GTK. Se espera que salga una versión con motor para soporte de Jython, con acceso a todas las librerías de java e incluyendo soporte para Swing y también para las librerías gráficas de python que ya mencionamos.

Zend Studio for Eclipse 6.0

Zend Studio es un entorno integrado de desarrollo para la programación en PHP. Forma parte, junto con Zend Platform de lo que Zend Technologies propone para el desarrollo Web, Zend Studio para la parte cliente y Zend Platform para la parte del servidor. Esta herramienta no requiere instalación previa de PHP, ni de ningún otro entorno de ejecución Java. Ha establecido el autocompletamiento de código, el plegado de código, inserción automática de paréntesis y corchetes, detección de errores de sintaxis en tiempo real y funciones de depuración. Posee también un manual de PHP integrado y un cliente FTP integrado.

Eclipse Editor PHP

Sin duda uno de los mejores entornos de desarrollo para PHP, Java, C/C++ y otros lenguajes, dispone de una potente administración de proyectos y ficheros, un gran editor con coloreado del lenguaje, detección y resaltado de errores sintácticos y de estructuras. Se trata de un entorno IDE cuya principal ventaja es la visualización de los errores de escritura y de inclusión de cabeceras, además dispone del manual de PHP integrado y de rápido acceso.

Para el desarrollo de la aplicación se decidió escoger el NetBeans IDE en su versión 6.7.1 como plataforma de desarrollo ya que permite crear aplicaciones Web con PHP y se encuentra disponible para los desarrolladores del centro como plataforma principal.

1.10.6 Protocolo XMPP

El *Extensible Messaging and Presence Protocol* (XMPP) es una tecnología para la comunicación en tiempo real. En esencia, el XMPP proporciona una vía para enviar piezas pequeñas de XML de una entidad a otra en tiempo real. Un servicio es una característica o función que puede ser usada por cualquier aplicación. XMPP típicamente proporciona los siguientes servicios:

- Codificación de canal

Este servicio, definido en [30], brinda encriptación de las conexiones entre un cliente y un servidor, o dos servidores para mayor seguridad de las aplicaciones.

- Autenticación

Este servicio, es definido para desarrollo de aplicaciones seguras. En este caso, el servicio de autenticación asegura que los entes intentando comunicarse deben ser primero autenticados por un servidor, que actúa como cierto portero para acceso de red.

- Presencia

La función de este servicio, definido en [31] es descubrir sobre la disponibilidad de red de otros entes. Un servicio de presencia responde la pregunta, ¿Está la entidad en línea y disponible para la comunicación o offline y no disponible? Los datos de presencia pueden también incluir información más detallada tal como si una persona está en una reunión. Típicamente, el compartir información de presencia está basado en un sistema de subscripción entre dos entes a fin de proteger la privacidad del usuario.

- *One-to-one messaging*

El propósito de este servicio es enviar mensajes a otra entidad. El uso clásico de uno a uno enviando mensajes que pueden ser arbitrarios XML, y cualesquiera dos entes en una red pueden intercambiar mensajes, ellos pueden ser servidores, componentes, dispositivos, servicios de red de XMPP habilitado, o cualquiera otra entidad de XMPP.

- Entorno *Peer-to-Peer* para sesiones media

Este servicio permite negociar y manejar una sesión de medios con otra entidad. La sesión puede ser usada para el propósito de charla de voz, charla de video, transferencia de archivo, y otras interacciones en tiempo real.

Existen otros servicios no menos importantes que para su profundización pueden ser analizados en "*XMPP: The Definitive Guide*" [32].

Una de las principales ventajas de este protocolo es que, a diferencia de otros protocolos de mensajería, se trata de un estándar libre. Una de sus extensiones (XEP-0060) está dedicada a la Publicación-Suscripción y es por tales razones por la que se ha elegido para la realización de este trabajo.

1.10.7 Protocolo BOSH

Bidirectional streams Over Synchronous HTTP (BOSH) es una pasarela para poder establecer una conexión a un servidor XMPP desde HTTP. Esto es posible porque XMPP establece una conexión persistente. Se mantiene la conexión desde el principio hasta el final, y mientras se envían todos los mensajes que quieran. BOSH es un estándar sumamente nuevo y se están creando librerías para su optimización. Actualmente brinda grandes facilidades para proyectos de tiempo real y es por ello que se decide poner en práctica para esta arquitectura.

- Puede transportar pequeñas instancias de XMPP sobre HTTP.
- Aumenta la escalabilidad de las aplicaciones usando XMPP.
- Los servidores de XMPP pueden manipular TONELADAS de usuarios concurrentes.
- Puede ser usado en situaciones dónde el servidor XMPP no está disponible.
- No pierde datos. Los mensajes se guardarán en el servidor hasta que el cliente esté nuevamente.
- BOSH define qué tan arbitrario pueden ser los elementos XML transportados eficazmente y de fuente fidedigna sobre el HTTP en ambas direcciones entre un cliente y servidor. [33]

1.10.8 Ventajas de la combinación XMPP/BOSH sobre marcos tradicionales web

Sin lugar a dudas la ventaja más evidente es una interacción en tiempo real. Esto no tiene directamente que ver con XMPP, sino que es proporcionada por el protocolo BOSH. Muchas técnicas similares han existido desde hace bastante tiempo, por ejemplo, el *comet* o *pushlets*. BOSH es

actualmente soportada por el servidor XMPP más conocidos, por lo que es natural para su uso con XMPP.

Es posible utilizar características importantes del servidor XMPP, o sea no hay necesidad de contar con bases de datos separadas de usuarios registrados y sus suscripciones, ya que esto estará gestionado por el servidor XMPP. Todos los cambios en la presencia del usuario (es decir, conectado / desconectado) y la presencia de componentes (reinicio y desconexión) también son atendidos. Es posible implementar fácilmente funciones de mensajería instantánea.

XMPP/BOSH también permite el estilo de mensajería asincrónica. Esto significa que el código que se envía en la solicitud no se haga a la espera de la respuesta, sino que la respuesta se controla mediante una función de devolución de llamada. En términos prácticos, esto hace que la aplicación web sea más ágil y fácil de usar. Técnicamente, el mismo comportamiento se puede implementar sin XMPP, es solo que la mensajería asincrónica es una forma natural de vida de XMPP, en lugar de un servidor web que está más acostumbrado a la secuencia "petición y respuesta".

1.10.9 Librerías

Strophejs 1.0.1

Strophejs [34] es una librería JavaScript dado que este último no facilita conexiones TCP persistentes, esta librería se basa en bidireccional flujos sobre HTTP sincrónico (BOSH siglas en inglés) para emular la persistencia, con estado de conexión, en ambos sentidos a un servidor XMPP.

Estrofa no solo es una API pequeña y fácil de entender, ya desde hace algunos años ha sido probado en los principales navegadores y su comunidad de desarrolladores ha ido aumentando para beneficio de los que se inician en su utilización. Strophe.Builder ayuda a construir estrofas rápidamente, algo muy importante en un clima actual donde la baja latencia es muy criticada por las personas. Esta librería detecta muchos errores e incluye optimizaciones que ningún otro marco de AJAX parece hacer ofreciendo así excelentes rendimientos. Estrofa es una excelente librería para implementar clientes XMPP, incluye robusto TLS y soporte SASL.

Sixties

Es una librería de clase PHP que pretende implementar la parte cliente de varias extensiones (XEP) del protocolo XMPP (Jabber). Consta de cuatro componentes: una librería para ampliar XMPPHP para

interactuar con el servidor de jabber PubSub; servicios Web REST a modo de interfaz; un robot que permite recibir notificaciones y realizar acciones de disparo; y una aplicación cliente a través de servicios web, que permite interactuar con los servidores Jabber (nodos administrados y suscripciones).

Las características de las librerías anteriormente descritas son de gran utilidad para la implementación del componente de comunicación en tiempo real. Con Strophejs se garantizan conexiones TCP persistentes para la parte del cliente y a través de Sixties se facilita la interacción del servidor XMPP con sus clientes ya que implementa el paradigma PubSub.

1.10.10 Servidor Ejabberd 2.1.5

Para la investigación se determinó como servidor XMPP al Ejabberd (*Erlang Jabber Daemon*). Es un servidor de mensajería instantánea de código abierto (GNU GPL) multiplataforma, distribuido y tolerante a fallos, características que hacen posible su elección. Esta versión actualmente es la más estable escrita principalmente en erlang y además está basada en los estándares abiertos para lograr la comunicación en tiempo real utilizando para ello el protocolo XMPP. A continuación se presentan algunas de las características más importantes del ejabberd.

Multiplataforma: Funciona en las plataformas operativas más populares.

Distribuido: Se puede ejecutar Ejabberd en más ordenadores y todos ellos se regirán por un único dominio de Jabber. Cuando deseas ampliar tu servidor Jabber, simplemente podrás agregar un nuevo nodo barato a tu clúster. De esta manera no necesitarás comprar una máquina potente y cara para que sea compatible con cientos de usuarios simultáneos.

Tolerante a fallos: Los nodos del clúster Ejabberd comparten algunas o todas las tablas de las bases de datos, para que toda la información requerida por un servicio que tiene que funcionar adecuadamente se almacene permanentemente no en más de un nodo. Si uno de los nodos da un fallo los otros seguirán funcionando sin ninguna interrupción. Además, también podrás añadir o reemplazar los nodos sobre la marcha.

Configuración sencilla: Ejabberd está construido con el lenguaje Erlang/OTP. No tienes que configurarlo en una base de datos externa ni en ningún otro servidor externo porque todo ya está instalado y listo para ejecutarse fuera del ordenador.

Hosts virtuales: Pueden alojarse diferentes hosts de Jabber en la misma instancia de Ejabberd. Es tan fácil como añadir un nuevo nombre de dominio a la lista de hosts del archivo de configuración.

Compatible con IPv6: Es compatible con IPv6 tanto para C2S como para S2S.

1.10.11 Servidor Web

Apache 2.0

El proyecto Apache comenzó en 1995 como un proyecto para la creación de un servidor Web de código abierto. Con los años se le añadieron muchas funcionalidades como la habilidad de un solo servidor de mantener múltiples sitios Web virtuales y esquemas de autenticación. Actualmente es uno de los servidores más usados para la creación de sitios Web comerciales. Está disponible para una gran cantidad de plataformas como GNU/Linux, Mac OS, Mac OS X Server, Netware, Open Step/Match, UNIX, Solaris, SunOS, UnixWare, Windows entre otras.

1.11 La evaluación arquitectónica, técnicas y métodos

El propósito de realizar evaluaciones a la arquitectura, es para analizar e identificar riesgos potenciales en su estructura y sus propiedades, que puedan afectar al sistema de software resultante, verificar que los requerimientos no funcionales estén presentes en la arquitectura, así como determinar en qué grado se satisfacen los atributos de calidad. Cabe señalar que los requerimientos no funcionales también son llamados atributos de calidad [35].

Un atributo de calidad es una característica de calidad que afecta a un elemento. Donde el término “característica” se refiere a aspectos no funcionales y el termino “elemento” a componente.

- Cuanto más temprano se encuentre un problema en un proyecto de software, mejor [36].
- Realizar una evaluación de la arquitectura es la manera más económica de evitar desastres.
- Analizar y evaluar la calidad exigida por los usuarios.
- Decisiones de diseño.
    - Restricciones de Implementación.
    - Fija la estructura organizacional, tanto del desarrollo, construcción y ejecución del sistema.
    - Logra los atributos de calidad.
    - Permite el prototipado.

  - Permite estimaciones más certeras.
- Abstracción transferible entre sistemas.

¿Cuándo una Arquitectura puede ser evaluada?

Es posible realizarla en cualquier momento según Kazman, pero propone dos variantes que agrupan dos etapas distintas: temprano y tarde [37].

- **Temprana**. No es necesario que la arquitectura esté completamente especificada para efectuar la evaluación, y esto abarca desde las fases tempranas de diseño y a lo largo del desarrollo.
- **Tarde**. Cuando ésta se encuentra establecida y la implementación se ha completado. Este es el caso general que se presenta al momento de la adquisición de un sistema ya desarrollado.

Reglas de oro para determinar el momento de realizar evaluaciones [34]:

1. Realizar una evaluación cuando el equipo de desarrollo inicia a tomar decisiones que afectan directamente a la arquitectura.
2. Cuando el costo de no tomar estas decisiones podría tomar más, que el costo de realizar una evaluación.

1.11.1 Técnicas de evaluación

Las técnicas utilizadas para la evaluación de atributos de calidad requieren grandes esfuerzos para crear especificaciones y predicciones. Entre las técnicas de evaluación de arquitecturas de software se destacan: evaluación basada en escenarios, evaluación basada en simulación, evaluación basada en modelos matemáticos y evaluación basada en experiencia. [36]

En el anexo 2 se muestran las técnicas de evaluación con sus respectivos instrumentos.

Evaluación basada en escenarios

Un escenario es una breve descripción de la interacción de alguno de los involucrados en el desarrollo del sistema. Un escenario consta de tres partes: el estímulo, el contexto y la respuesta. El estímulo es la parte del escenario que explica o describe lo que el involucrado en el desarrollo hace para iniciar la interacción con el sistema. Puede incluir ejecución de tareas, cambios en el sistema, ejecución de pruebas y configuración. El contexto describe qué sucede en el sistema al momento del estímulo. La

respuesta describe, a través de la arquitectura, cómo debería responder el sistema ante el estímulo. Este último elemento es el que permite establecer cuál es el atributo de calidad asociado.

Entre las ventajas de su uso están se pueden ver que son simples de crear y entender, son poco costosos y efectivos.

Actualmente las técnicas basadas en escenarios cuentan con dos instrumentos de evaluación relevantes, a saber: el *Utility Tree* propuesto por Kazman [38], y los *Profiles*, propuestos por Bosch [39].

Utility Tree

Es un esquema en forma de árbol que presenta los atributos de calidad de un sistema de software, refinados hasta el establecimiento de escenarios que especifican con suficiente detalle el nivel de prioridad de cada uno. La intención del uso del Utility Tree es la identificación de los atributos de calidad más importantes para un proyecto particular. El Utility Tree contiene como nodo raíz la utilidad general del sistema. Los atributos de calidad asociados al mismo conforman el segundo nivel del árbol los cuales se refinan hasta la obtención de un escenario lo suficientemente concreto para ser analizado y otorgarle prioridad a cada atributo considerado. Cada atributo de calidad perteneciente al árbol contiene una serie de escenarios relacionados, y una escala de importancia y dificultad asociada, que será útil para efectos de la evaluación de la arquitectura.

Para especificar la calidad se hace uso de un árbol de utilidad, el cual tiene en la raíz la bondad o utilidad del sistema, en el segundo nivel del árbol se encuentran los atributos de calidad y las hojas del árbol son los escenarios, los cuales representan mecanismos mediante los cuales extensas declaraciones de cualidades son hechas específicas y posibles de evaluar. La generación del árbol de utilidad permite establecer prioridades.

Perfiles (profiles)

Un perfil es un conjunto de escenarios, generalmente con alguna importancia relativa asociada a cada uno de ellos. El uso de perfiles permite hacer especificaciones más precisas del requerimiento para un atributo de calidad. Para la definición de un perfil, es necesario seguir tres pasos: definición de las categorías del escenario, selección y definición de los escenarios para la categoría y asignación del "peso" a los escenarios. Cada atributo de calidad tiene un perfil asociado.

Evaluación basada en modelos matemáticos

La evaluación basada en modelos matemáticos se utiliza para evaluar atributos de calidad operacionales, permite una evaluación estática de los modelos de diseño arquitectónico, se presentan como alternativa a la simulación, dado que evalúan el mismo tipo de atributos. Ambos enfoques pueden ser combinados, utilizando los resultados de uno como entrada para el otro. Entre los instrumentos que se cuentan para las técnicas de evaluación de arquitecturas de software basada en modelos matemáticos, se encuentran las Cadenas de Markov y los *Reliability Block Diagrams*.

Entre las desventajas que presenta esta técnica se encuentra la inexistencia de modelos matemáticos apropiados para los atributos de calidad relevantes y el hecho de que el desarrollo de un modelo de simulación completo puede requerir esfuerzos sustanciales.

Evaluación basada en experiencia

Existen dos tipos de evaluación basada en experiencia: la evaluación informal, que es realizada por los arquitectos de software durante el proceso de diseño, y la realizada por equipos externos de evaluación de arquitecturas.

Evaluación basada en simulación

En el ámbito de las simulaciones, se encuentra la técnica de implementación de prototipos (*prototyping*). Esta técnica implementa una parte de la arquitectura de software y la ejecuta en el contexto del sistema. Es utilizada para evaluar requerimientos de calidad operacional, como desempeño y confiabilidad. Para su uso se necesita mayor información sobre el desarrollo y disponibilidad del hardware, y otras partes que constituyen el contexto del sistema de software. Según Bosch se obtiene un resultado de evaluación con mayor exactitud.

La exactitud de los resultados de la evaluación depende, a su vez, de la exactitud del perfil utilizado para evaluar el atributo de calidad y de la precisión con la que el contexto del sistema simula las condiciones del mundo real.

1.11.2 Métodos de evaluación

Cuando se trata de garantizar calidad en una etapa temprana, un punto de partida importante es la arquitectura del software. Un método de evaluación sirve de guía a los involucrados en el desarrollo

del sistema, en la búsqueda de conflictos que puede presentar una arquitectura, y sus soluciones. Por esta razón, resulta conveniente estudiar los métodos de evaluación de arquitecturas de software propuestos hasta el momento. [40]

Software Architecture Analysis Method (SAAM)

Según Kazman [37], el Método de Análisis de Arquitecturas de Software (*Software Architecture Analysis Method*, SAAM) es el primero que fue ampliamente promulgado y documentado. El método fue originalmente creado para el análisis de la modificabilidad de una arquitectura, pero en la práctica ha demostrado ser muy útil para evaluar de forma rápida distintos atributos de calidad, tales como modificabilidad, portabilidad, escalabilidad e integrabilidad.

Con la aplicación de este método, si el objetivo de la evaluación es una sola arquitectura, se obtienen los lugares en los que la misma puede fallar, en términos de los requerimientos de modificabilidad. Para el caso en el que se cuenta con varias arquitecturas candidatas, el método produce una escala relativa que permite observar qué opción satisface mejor los requerimientos de calidad con la menor cantidad de modificaciones.

Architecture Trade-off Analysis Method (ATAM)

El método se concentra en la identificación de los estilos arquitectónicos o enfoques arquitectónicos utilizados. Estos elementos representan los medios empleados por la arquitectura para alcanzar los atributos de calidad, así como también permiten describir la forma en la que el sistema puede crecer, responder a cambios, e integrarse con otros sistemas, entre otros [36].

Lo que se pretende hacer con ATAM a demás de mejorar la documentación, es registrar los posibles *riesgos*, los *no riesgos*, los *puntos de sensibilidad* y los *puntos de desventajas* que encontramos en el análisis de la arquitectura.

El método de evaluación ATAM comprende nueve pasos, agrupados en cuatro fases (Presentación, Investigación y Análisis, Pruebas y Reporte). El anexo 3 muestra las fases.

Active Reviews for Intermediate Designs (ARID)

De acuerdo con Kazman el método ARID es conveniente para realizar la evaluación de diseños parciales en las etapas tempranas del desarrollo. El método ARID consta de nueve pasos recogidos

en dos fases (Actividades Previas y Evaluación).

ARID es un método de bajo costo y gran beneficio, es un híbrido entre ADR y ATAM [41]. Se basa en ensamblar el diseño de los stakeholders para articular los escenarios de usos importantes o significativos, y probar el diseño para ver si satisface los escenarios.

La tabla del anexo 4 muestra una comparación entre los métodos estudiados: SAAM, ATAM y ARID analizando aspectos relevantes como los atributos de calidad que contempla cada uno, los objetivos que analizan, las etapas o fases del proyecto en las que se aplican, así como sus principales enfoques.

Después de haberse realizado un estudio de las técnicas y métodos de evaluación de la arquitectura se propone realizar el proceso utilizando como técnica la basada en escenarios. Esta técnica permite identificar el estimulo, el contexto y la respuesta del sistema asociados a un atributo de calidad determinado. Su instrumento de evaluación escogido fue el *Utility Tree* como elemento principal de identificación de los atributos de calidad más importantes así como sus prioridades. El método seleccionado fue ATAM por facilitar identificar los posibles riesgos, los no riesgos, los puntos de sensibilidad y los puntos de desventajas que se encuentren en el análisis de la arquitectura.

Conclusiones parciales del capítulo

Este capítulo es el resultado de la búsqueda y el análisis de la información vinculada al objeto de estudio (AS) y los sistemas existentes asociadas al campo de acción. Después de haber realizado el estudio de cada uno de estos temas y teniendo en cuenta las características del componente fue seleccionado el estilo de arquitectura en capas en conjunto con el estilo basado en eventos de la familia de estilos *Peer-To-Peer*. Como metodología de desarrollo se seleccionó OpenUP, utilizando UML como lenguaje de modelado, y Visual Paradigm 6.1 como herramienta CASE. Como lenguaje de programación se escogió PHP y como IDE de desarrollo al NetBeans 6.7.1 por su integración con subversión y el lenguaje seleccionado. Se propuso utilizar como servidor XMPP al servidor Ejabberd en su versión 2.1.5 por las características que posee para la comunicación en tiempo real. El servidor web será apache 2.0 y se proponen las librerías Dojo 1.3, Sixties y Strophejs 1.0.1.

CAPÍTULO 2: DESCRIPCIÓN DE LA ARQUITECTURA

Introducción

A lo largo de este documento se ha hecho referencia a distintos elementos que han de colaborar para llevar a feliz término el proceso de construcción del diseño arquitectónico. El presente capítulo se dedica a la organización del sistema, los requerimientos de hardware y esencialmente a la representación del funcionamiento del sistema a través de las vistas arquitectónicas.

2.1 Organización del sistema

En la figura 4 se muestra la forma de organización del sistema para ofrecer una mejor comprensión de la arquitectura propuesta.

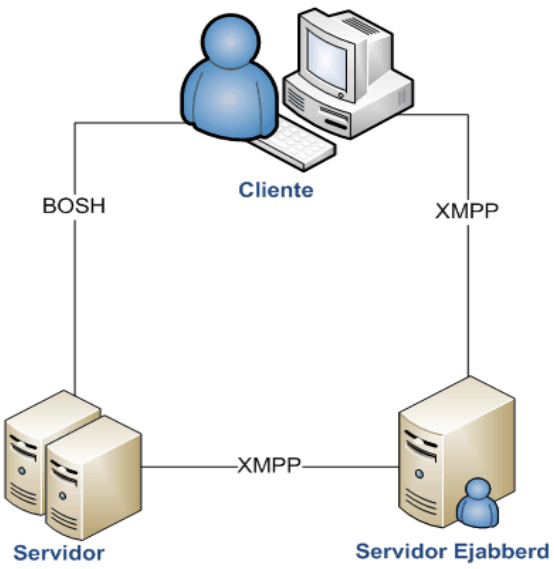

Figura 4: Organización del sistema.

Como se observa en la figura el sistema contará con un servidor de aplicaciones, un servidor Ejabberd (XMPP) y una o varias computadoras (PC) clientes. El protocolo BOSH es el encargado de transportar los datos de manera eficiente y con una latencia mínima en ambas direcciones cliente-servidor. Para aplicaciones que requieren tanto de empuje y atracción semántica, BOSH reduce significativamente el consumo del ancho de banda que otros protocolos y técnicas actuales comúnmente conocidas como Ajax. Como se observa el sistema constará de un servidor Ejabberd, entidad que contiene los nodos a

través del cual los productores publican la información y se realizan las peticiones de suscripción por los consumidores interesados en conocer la información publicada.

En la organización del sistema se abordan una serie de funcionalidades y elementos que se requieren para el correcto funcionamiento del software. No menos importante son los requerimientos de hardware, y es por ello que en esta investigación el siguiente punto es dedicado a este tema.

2.2 Metas y restricciones arquitectónicas

Los requisitos no funcionales son aquellas propiedades que debe tener la solución y que no son una funcionalidad [12], como por ejemplo: alta disponibilidad, flexibilidad, interoperabilidad, mantenimiento, rendimiento, fiabilidad, reusabilidad, escalabilidad, seguridad, robustez entre otros. Dentro de las metas y restricciones arquitectónicas definidas para el componente, se encuentran los requisitos no funcionales que se enuncian a continuación.

RNF 1. Apariencia o interfaz externa

- RNF 1.1 El sistema debe estar diseñado de forma tal que el usuario interactúe sin dificultad alguna con la aplicación, ajustándose a los estándares establecidos para el desarrollo de un buen diseño.

RNF 2. Usabilidad

- RNF 2.1 La aplicación tiene que ser capaz de ofrecer facilidades de uso para un buen entendimiento y aceptación del producto por los usuarios finales.
- RNF 2.2 El sistema podrá ser usado por cualquier tipo de persona que posea conocimientos básicos en el manejo de la computadora y el trabajo en la web.

RNF 3. Soporte

- RNF 3.1 El sistema debe propiciar su mejoramiento y la incorporación de otras opciones en el futuro.

RNF 4. Portabilidad

- RNF 4.1 El sistema debe ser ejecutado sobre el sistema operativo Linux, por su característica de ser código abierto.

RNF 5. Seguridad

- RNF 5.1 El sistema debe establecer un mecanismo que permita la utilización de los servicios por usuarios autorizados.
- RNF 5.2 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.

RNF 6. Fiabilidad

- RNF 6.1 El sistema debe ser confiable y preciso en la información que le suministra al usuario para evitar cualquier tipo de error.
- RNF 6.2 El sistema estará disponible todo el tiempo, permitiendo el trabajo de los usuarios y las acciones de mantenimiento.

RNF 7. Ayuda

- RNF 7.1 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.

RNF 8. Disponibilidad

- RNF 8.1 El sistema debe permanecer en constante funcionamiento, interrumpiendo el mismo solo cuando sea necesario reiniciarlo o detenerlo por tareas de mantenimiento o cambio de la configuración.

RNF 9. Software

- RNF 9.1 El software base debe responder a las políticas de Software Libre y de Fuente Abierta.
- RNF 9.2 Se debe disponer del sistema operativo Linux y del navegador Mozilla Firefox 3.6.12 o superior.
- RNF 9.3 Se debe instalar el servidor Ejabberd en su versión 2.1.5.

RNF 10. Rendimiento

- RNF 10.1 El intercambio de comandos, eventos y estados de la comunicación debe ser en tiempo real.

RNF 11. 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.
- 1 GB de RAM.
- 80 GB de capacidad en disco.
- Tarjeta de red.

Servidor de Aplicaciones

- Procesador Pentium 4 a 3.00 GHz o superior.
- 2 GB de memoria RAM.
- 250 GB de capacidad en disco.
- Tarjeta de red.

Servidor Ejabberd

- Procesador Pentium 4 a 3.00 GHz o superior.
- 4 GB de memoria RAM.
- 250 GB de capacidad en disco.
- Tarjeta de red.

2.3 Vistas arquitectónicas

La arquitectura es un conjunto organizado de elementos, se utiliza para especificar las decisiones estratégicas acerca de la estructura y funcionalidad del sistema, las colaboraciones entre sus distintos elementos y su despliegue físico para cumplir las responsabilidades bien definidas.

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: describe los aspectos de concurrencia y sincronización del diseño.
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. [42]

El objetivo de las vistas de la arquitectura es la simplificación o abstracción de los modelos, de los cuales se destacan los detalles más significativos y se obvian los menos. En el presente documento no se describirá la vista de proceso por no representar información relevante para la descripción de la arquitectura propuesta ya que esta vista se centra en describir aspectos de concurrencia que son implementados por el servidor Ejabberd y no es objetivo de este trabajo describir los procesos internos de tal servidor, simplemente hacer uso del mismo.

2.3.1 Vista de Casos de Uso

Los diagramas de casos de uso muestran un conjunto de casos de uso, actores y sus relaciones. Se emplean para modelar la vista de casos de uso de un sistema. Esta vista cubre principalmente el comportamiento del sistema, como los servicios visibles externamente que proporciona el sistema en el contexto de su entorno [43]. La confección de este diagrama facilita la comprensión y garantiza una fácil asimilación para aquellas personas que desconocen del proyecto, además de que representa gráficamente la situación que posteriormente será analizada y diseñada.

La vista en cuestión de esta investigación contiene los casos de usos arquitectónicamente significativos para el componente de comunicación.

Como se muestra en la figura 5 el componente cuenta con dos actores, el Usuario y el Sistema externo. El Usuario representa el consumidor que hará uso del sistema y visualizará los datos en tiempo real. El Sistema externo es el encargado de servir los datos al usuario a través del componente de comunicación en tiempo real.

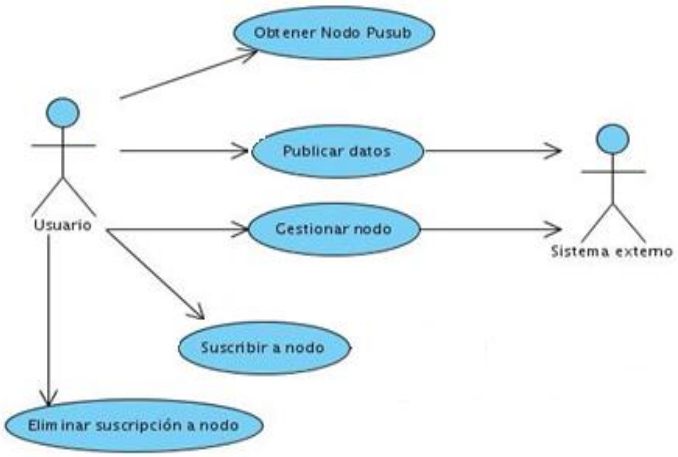

Figura 5. Diagrama de Casos de Uso

A continuación se verá el proceso que describen los casos de uso (CU).

CU Obtener Nodo Pubsub: Permite al usuario conectado solicitar el listado de los nodos pubsub existentes en el servidor Ejabberd.

CU Suscribir a Nodo: Permite al usuario realizar la suscripción a un nodo pubsub, por lo que se necesita, el nombre del servicio pubsub y el nombre del nodo, para comenzar a recibir los datos que sean publicados en el mismo.

CU Gestionar Nodo: Permite al usuario crear un nodo, para lo cual introduce los datos correspondientes, el sistema guarda los datos y crea el nodo, en caso de que necesita introducir algún cambio pues edita la información del nodo y el sistema guarda los cambios, en caso de que fuera a eliminar el sistema muestra el listado de nodos registrados para proceder con la operación.

CU Publicar Datos: Es la acción del sistema externo de publicar datos en tiempo real en el nodo pubsub al que el usuario suscribió.

CU Eliminar suscripción a nodo: Permite al usuario eliminar la suscripción existente de forma que ya no se reciban más datos del destino al que previamente se suscribió.

2.3.2 Vista Lógica

La vista Lógica representa los elementos de diseño más importantes para la arquitectura del sistema. Representa un conjunto arquitectónicamente significativo de paquetes de alto nivel, subsistemas de diseño e interfaces.

Consiste en mostrar una propuesta de subsistemas en los que se dividirá la aplicación. En algunas ocasiones se tiene la necesidad de organizar los elementos de un diagrama en un grupo. Generalmente se pretende mostrar ciertas clases o componentes que son parte de un subsistema en particular. Básicamente un paquete agrupa los elementos de un diagrama.

La vista lógica del componente *Dashboard* se representa a través de tres capas o paquetes fundamentales en la figura 6 denominados: Presentation, App y Data Access. A continuación se presenta la descripción de las capas que conforman la vista lógica.

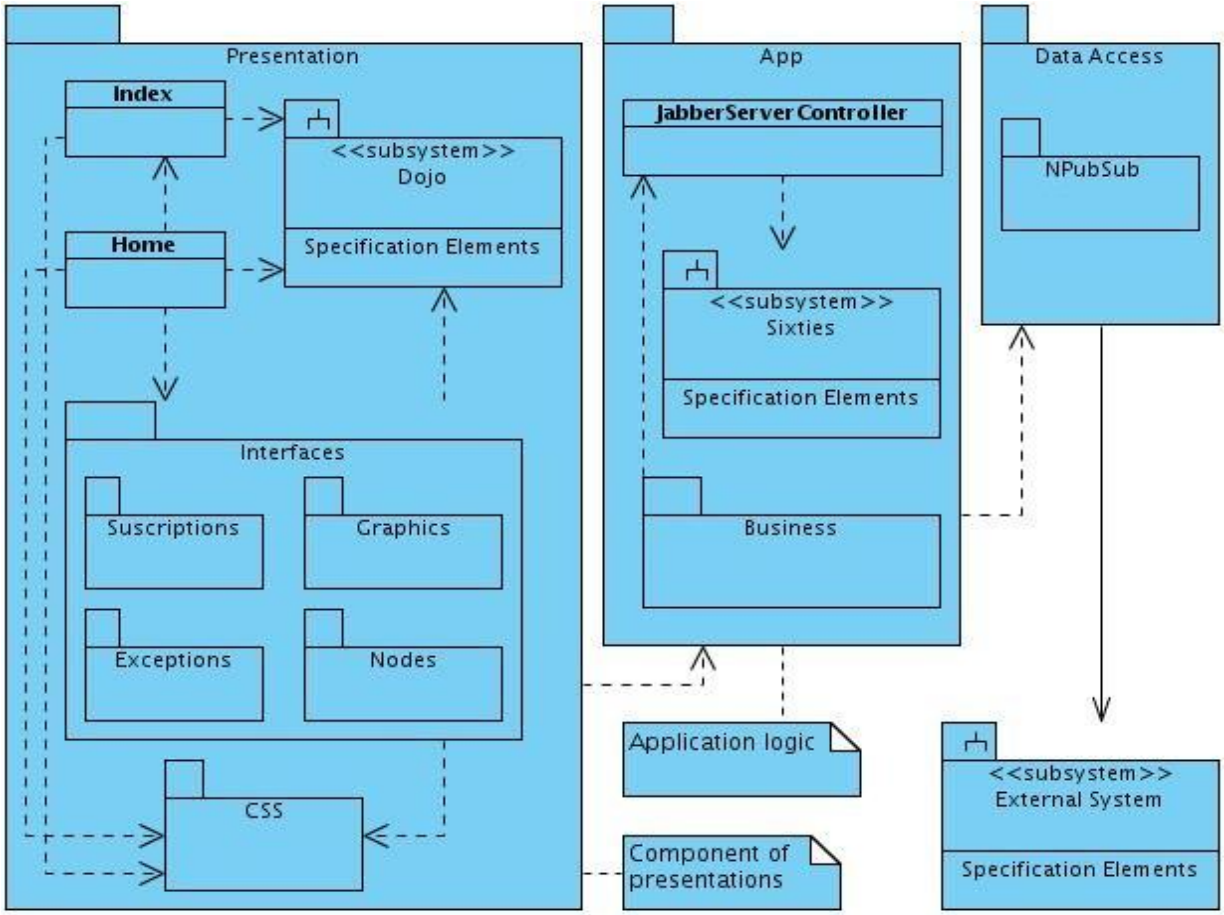

Figura 6: Vista Lógica General

Capa Presentation

Esta es la capa que interactúa con el usuario final, es la encargada de presentar la información y recolectarla para hacer uso de los servicios expuestos por la capa de aplicación **App**, para satisfacer los casos de uso de la aplicación.

El Paquete Interfaces: es el paquete que contiene los sub-paquetes Nodes, Graphics, Suscriptions y Exceptions. El Paquete Graphics: contiene los gráficos a través de los cuales se visualizarán los datos. El Subsistema Dojo: para mejorar la usabilidad se utiliza el Framework en JavaScript Dojo el cual permite añadir características dinámicas fácilmente a las páginas Web. El Paquete Suscriptions: contiene las vistas de las suscripciones. El Paquete Nodes: contiene las vistas que gestionan las acciones sobre los nodos. El Paquete Exceptions: Contiene las vistas que lanzan las excepciones de funcionalidad y de una autenticación inválida. El Componente Index: Es la vista encargada de la autenticación de los usuarios. El Componente Home: Vista encargada de mostrar las funcionalidades del sistema al usuario.

Capa App

Esta capa contiene todas las clases que modelan el dominio (negocio), sus entidades y sus relaciones. El Paquete Business: contiene las clases que modelan el dominio. El Subsistema Sixties: es el subsistema que implementa las funcionalidades que permiten interactuar con la capa Data Access.

Capa Datos Access

Esta capa contiene el paquete NPubSub que se encarga de la publicación y subscripción en el servidor Ejabberd. Se representa además el subsistema de implementación External System, al cual se le integra el componente de comunicación para mostrar los datos en tiempo real. Lo determinará una fuente de datos que necesite de la reutilización del componente definido en esta investigación.

En este esquema en tres capas la lógica de aplicación queda en la capa de dominio (capa App). De la lógica de aplicación se encarga el componente JabberServerControler que hace uso del subsistema Sixties para implementar debidamente el funcionamiento de la aplicación. Del dominio se encarga el componente Business. Aún estando la lógica de aplicación y la lógica de dominio en la misma capa, el

modelo cuenta con un buen desacoplamiento debido a que internamente se encuentran independientes una de la otra facilitando así, su mantención y reutilización.

2.3.3 Vista de 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 de software de un sistema contiene los nodos que forman la topología hardware sobre la que se ejecuta el sistema. Se preocupa principalmente de la distribución, entrega e instalación de las partes que constituyen el sistema. Los aspectos estáticos de esta vista se representan mediante los diagramas de despliegue [43, pág. 25]. El sistema estará distribuido como se muestra en la figura 7.

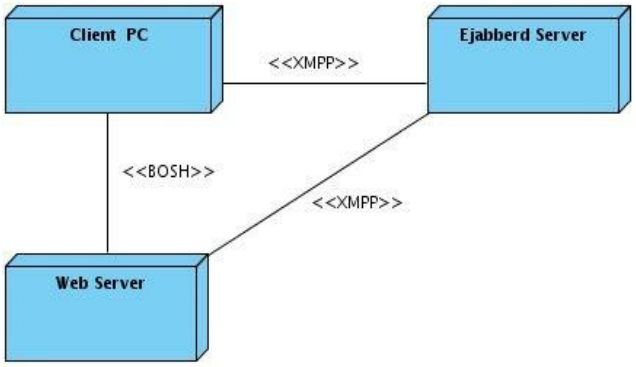

Figura 7: Diagrama de despliegue.

La aplicación será desplegada de la siguiente manera: un cliente (Cliente PC) que podrá estar en cualquier parte utiliza el protocolo BOSH para emular una conexión persistente, con estado y conexión bidireccional al servidor Ejabberd (nodos administrados y suscripciones). El servidor Ejabberd a través del protocolo de comunicación XMPP es el encargado de gestionar las conexiones y los envíos de paquetes XML entre el cliente y los servidores. El servidor web es el encargado de gestionar la distribución de las operaciones que deben realizarse para obtener el resultado.

De esta forma, se garantizará el funcionamiento del componente, y el despliegue del mismo en cualquier entorno que cuente con una red local o acceso a Internet.

2.3.4 Vista de Implementación

La vista de implementación es representada por un diagrama de componentes o especificaciones de paquetes. En la vista de componentes [43, pág. 82] de un sistema se representa la organización y las

dependencias que existen entre los componentes que se van a utilizar para dar solución al problema. Un componente es un modulo software del sistema o un subsistema por si mismo (código fuente, código binario o código ejecutable), contiene la implementación de una o varias clases y puede tener varias interfaces. Las interfaces representan la parte del componente que es pública y puede ser utilizada por otros componentes.

En los diagramas de componentes pueden aparecer paquetes que representan un conjunto de componentes relacionados lógicamente y suelen coincidir con los paquetes lógicos. Estos paquetes permiten una división física de nuestro sistema. Los subsistemas son una colección de componentes y otros subsistemas de implementación usados para estructurar el modelo de implementación y dividirlos en pequeñas partes. El diagrama de componente se estructuró como se muestra en la figura 8 en componentes y subsistemas.

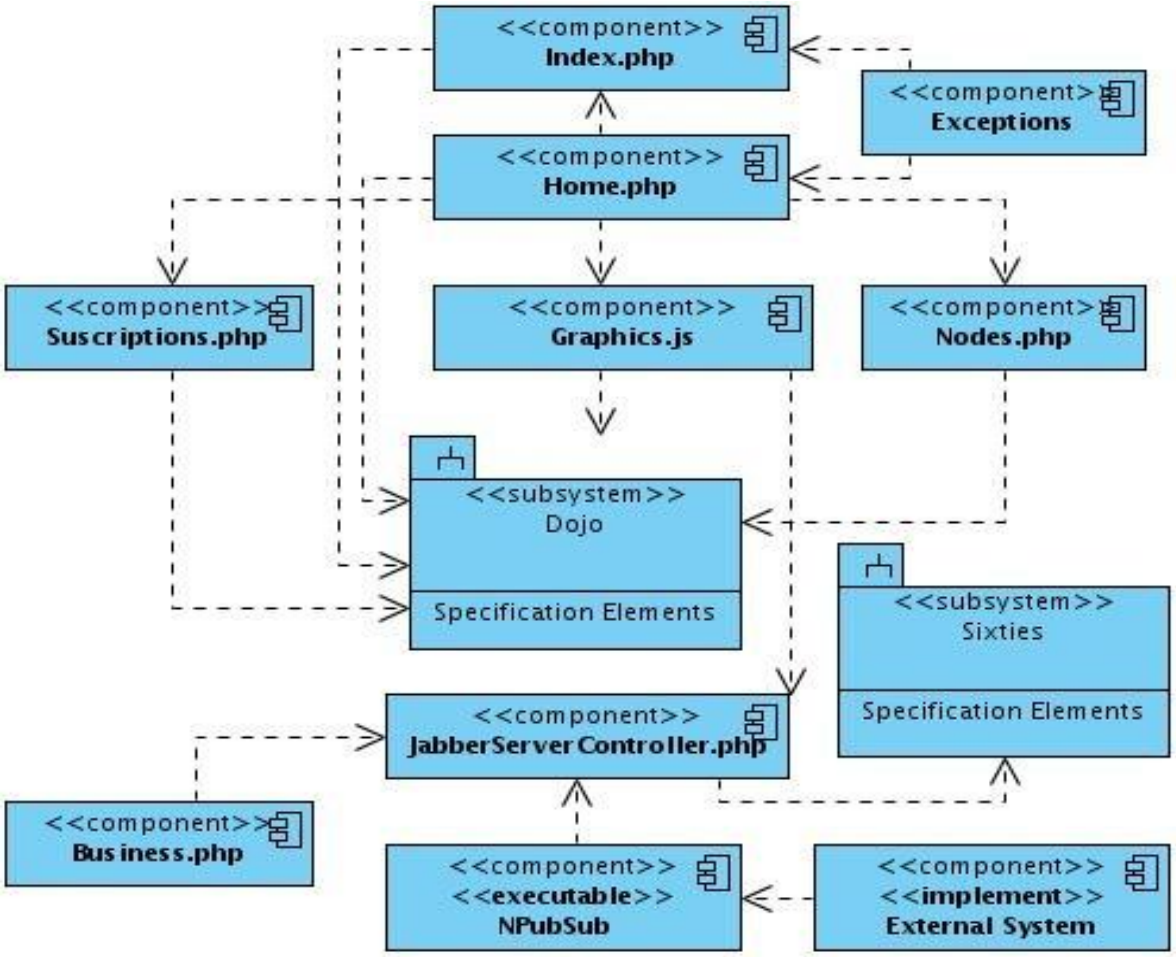

Figura 8: Diagrama de Componentes

Todos estos componentes son importantes ya que representan una parte funcional del sistema. A través del componente Home se realizan todas las acciones de la aplicación. El componente Suscriptions es el encargado de las funcionalidades referentes a los casos de uso Suscribir a Nodo y Eliminar Suscripción, así el componente Nodes se encarga de permitir a los usuarios la gestión de los nodos. El componente Graphics se encarga de visualizar los gráficos y para que estos entiendan la instantaneidad, se utiliza la librería Strophejs que implementa los servicios del protocolo BOSH para facilitar conexiones TCP persistentes. Ver Figura 9

Con el Subsistema Dojo se aplica el dinamismo a las diferentes vistas y mediante él se realiza la visualización de los gráficos en tiempo real.

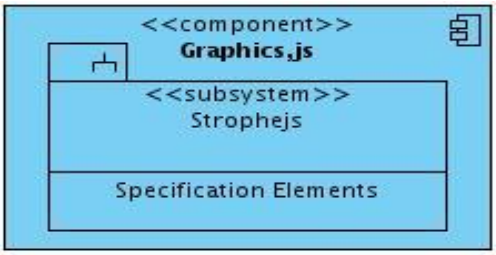

Figura 9 Componente Gráficos

El componente JabberServerControler se encarga de toda la lógica de la aplicación y se auxilia del subsistema Sixties para la implementación del paradigma PubSub, ver Figura 10. El Componente Business encierra todo el dominio referente a la aplicación en cuestión. El componente NPubSub es a través del cual se realiza la publicación-suscripción en el servidor Ejabberd, servidor que soporta XMPP como protocolo para la comunicación de los datos en tiempo real. Esta arquitectura no se define como suelen realizarse normalmente con un Sistema Gestor de Base de Datos ya que el Ejabberd incluye una base de datos internamente y se encarga de realizar las tareas referentes al manejo de los nodos, suscripciones y datos.

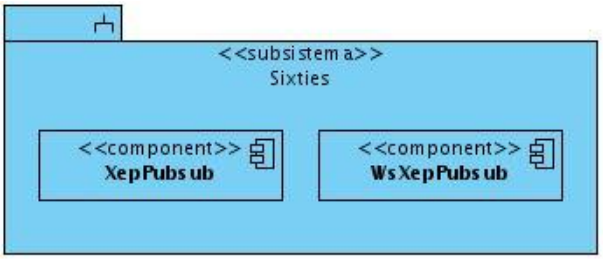

Figura 10 Subsistema Sixties

Distribución Física de los Componentes

En la figura 11 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:

Sixties: Permite la comunicación en tiempo real entre los servicios web y el servidor Ejabberd.

Ejabberd: Permite las suscripciones a un nodo así como publicar la información y enviar notificaciones de eventos a todas las entidades clientes que están autorizadas para conocer la información publicada.

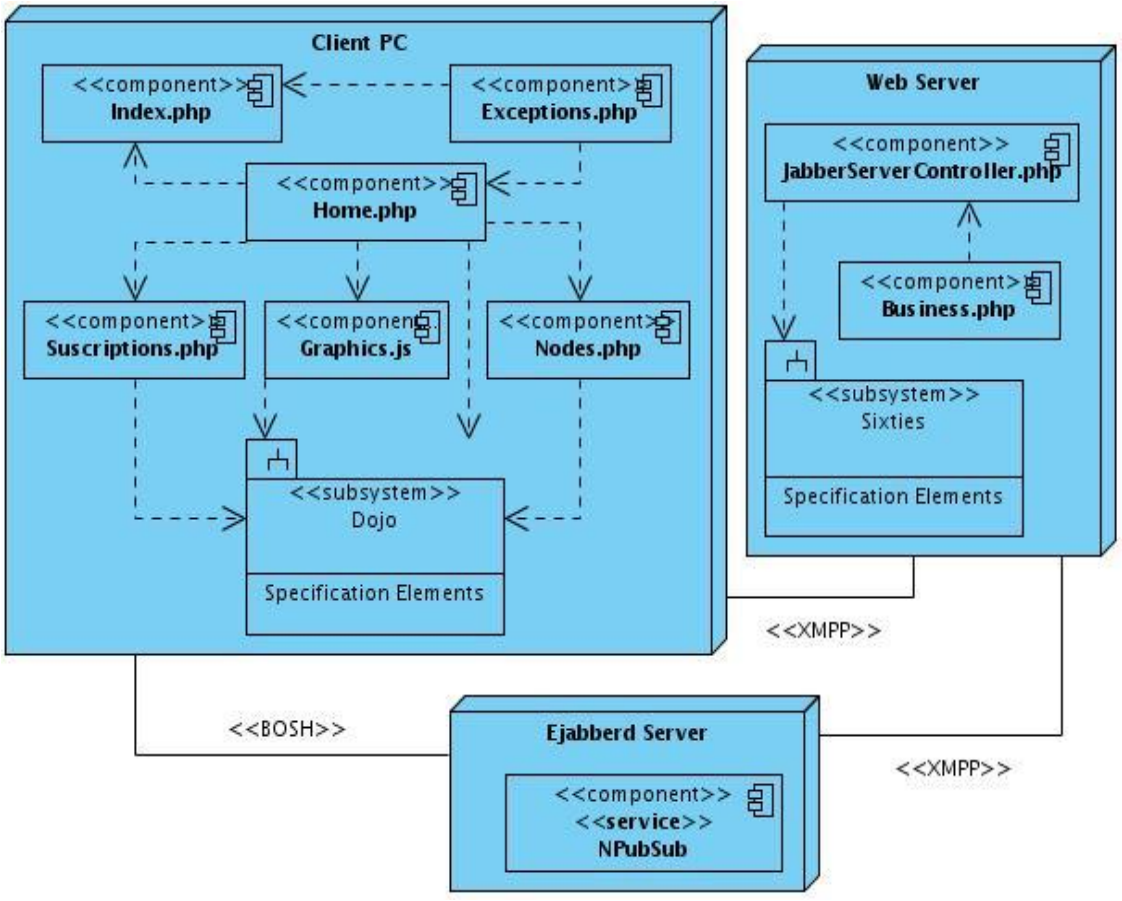

Figura 11 Distribución Física de los Componentes

Conclusiones parciales del capítulo

En el capítulo se propone de la organización del sistema, así como los principales objetivos, mecanismos y restricciones arquitectónicas. Se realizó la descripción de la arquitectura del sistema a través de las 4+1 vistas de la arquitectura definidas por Phelipe Kruchten. Las vistas arquitectónicas permitieron representar la propuesta de solución del sistema desde diferentes perspectivas, proporcionando a los desarrolladores una visión común del mismo.

CAPÍTULO 3. EVALUACIÓN DEL DISEÑO ARQUITECTÓNICO PROPUESTO

Introducción

Evaluar una AS previene 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. En el presente capítulo se realiza el proceso de evaluación de la AS a través del método ATAM, dando lugar a los resultados arrojados por dicha tarea.

3.1 Proceso de evaluación de la arquitectura

Para la evaluación de la arquitectura del componente de comunicación se tomaron en cuentas los atributos de calidad relacionados directamente con la arquitectura del software: seguridad, rendimiento, funcionalidad, eficiencia, usabilidad, mantenibilidad, definidos por la ISO/IEC 9126 [44] e incluso se incorporaron otros que también se consideraron importantes desde el punto de vista arquitectónico. Además se decidió de las técnicas estudiadas seguir la línea de la técnica basada en escenarios con el instrumento de evaluación el Utility Tree, independientemente de que van implícitos en el método basado en arquitectura ATAM seleccionado para realizar la evaluación, el cual apoya a los involucrados con el proyecto a entender las consecuencias de las decisiones arquitectónicas respecto a los atributos de calidad del sistema.

3.2 Priorización de los escenarios de calidad

Se debe priorizar y ordenar los escenarios ya que así los arquitectos podrán contar con más orientación para tomar decisiones arquitectónicas, y los stakeholders pueden estar más conscientes de lo que esperan del sistema, y obtener una idea sobre en qué medida se va a sentir satisfecho. Una vez que se ha decidido incluir en el árbol de utilidad los escenarios más importantes, se procede a asignar un orden a los escenarios de calidad de un sistema utilizando dos criterios, a saber:

a) Evaluar el riesgo de no considerar el escenario. Se deben responder las preguntas: ¿qué pasa si este escenario no se cumple?, ¿cuántas personas se ven afectadas?, ¿es posible compensar el no responder a este escenario?
b) Evaluar el esfuerzo necesario para lograr cumplir con el escenario. En este caso se determina que cambios o integraciones de nuevos componentes se deben realizar para satisfacer el escenario. Los escenarios más difíciles son aquellos en los que se debe consumir más tiempo de análisis y el resto debe ser guardado como parte del registro.

Luego se construye el árbol de utilidad, la prioridad del escenario define en este método como un par (X, Y) en el cual X define el esfuerzo de satisfacer el escenario, y la Y indica los riesgos que se corren al excluirlos del árbol de utilidad. Los posibles valores para X e Y son A (Alto), B (Bajo) y M (Medio). El árbol de utilidad generado se toma como un plan para el resto de la evaluación que realiza el método. Indica además al equipo evaluador dónde ocupar su tiempo y dónde probar aproximaciones y riesgos arquitectónicos.

Árbol de Utilidad			
Atributo de Calidad	**Sub-característica**	**Escenario**	**Prioridad**
Funcionalidad	Seguridad	El sistema debe restringir el acceso a los datos	A
		El sistema mediante el servicio PubSub determina los privilegios a los clientes y aplicación que están autorizados a publicar y suscribirse.	A
		El sistema debe aceptar los datos introducidos por el usuario solo cuando estos sean los correctos.	A
	Adecuación	El sistema cumple con los RF y RNF solicitados por el cliente.	A
	Interoperabilidad	Debe ser capaz de integrarse a otros sistemas, permitiendo la comunicación en tiempo real.	M
Fiabilidad	Tolerancia a fallos y Recuperación	Al ocurrir un error en el servicio PubSub, la operación queda cancelada por lo que se le envía un mensaje al usuario y el sistema deberá retornar a su estado inicial antes de iniciada la petición.	A
Usabilidad	Comprensibilidad, Aprendibilidad y Atractividad	El sistema debe tener la capacidad de ser atractivo, entendible para el usuario.	M
Eficiencia	Usabilidad de Recursos y Conectividad	El sistema debe ser capaz de actualizar bidireccionalmente la información manejada durante el período en que no se mantuvo la conexión.	A
	Tiempo de Respuesta	Los tiempos de respuesta a las peticiones realizadas por los usuarios deben ser en tiempo real.	A
Mantenibilidad	Cambiabilidad y Facilidad de adaptación al cambio	El sistema debe permitir que se le puedan adicionar componentes, módulos o nuevas funcionalidades.	M
	Facilidad de adaptación al cambio	Los componentes pueden instalarse fácilmente en todos los ambientes para su	M

		funcionamiento.	
Portabilidad		Es fácil instalar las actualizaciones del sistema.	M
Leyenda: Prioridad Alta (A), Media (M), Baja (B)			

Tabla 1 Árbol de Utilidad DahBoard

ATAM es un método para identificar riegos, esto significa que se detecta áreas de riesgos potenciales dentro de la arquitectura de un complejo sistema de software. Por lo que ATAM tiene algunas implicaciones:

- Debe ser ejecutado tempranamente en el ciclo de desarrollo del software.
- Debe ejecutarse relativamente con un bajo costo y rápido, ya que evalúa los artefactos del diseño arquitectónico.
- Produce un análisis en proporción con el nivel de detalle de la especificación de la arquitectura. Además no siempre cuando se obtiene un análisis detallado de los atributos de calidad medibles de un sistema puede ser el éxito. En cambio el éxito es lograr identificar las amenazas potenciales para el sistema.

Este último aspecto es crucial para entender los objetivos de ATAM, donde el objetivo no es predecir exactamente el comportamiento de un atributo de calidad. Esto será imposible en etapas tempranas en el escenario de diseño, porque todavía no se tiene suficiente información para hacer esta predicción. Es por ello que ATAM se centra en la identificación de riesgos. Es sumamente importante en ATAM registrar cualquier riesgo, punto sensible y puntos de intercambio.

Los riesgos son decisiones arquitecturalmente importantes, que no han sido tomadas o decisiones que han sido tomadas pero las consecuencias no han sido entendidas a plenitud.

Los puntos sensibles son parámetros en la arquitectura en los cuales la respuesta medible de algunos atributos de calidad es altamente correlacionada.

Un punto de intercambio (*tradeoff*) es descubierto en la arquitectura cuando un parámetro de construcción arquitectural es un anfitrión para más de un punto sensible donde los puntos de calidad medibles son afectados indistintamente por cambio en el parámetro.

Las tablas desde la 2 a la 13 muestran el tratamiento de los escenarios a través del método ATAM como parte del proceso de evaluación de la arquitectura. Cabe descartar la simbología utilizada en las

tablas.

S es un punto sensible, como R es un riesgo, NR no riesgo y T es un punto de intercambio o *Tradeoff.* Estos contendrán un número que se incrementarán en la medida que se van identificando algunas de estas características.

Escenario #: 1	**El sistema debe restringir el acceso a los datos.**			
Atributo(s)	Funcionalidad-Seguridad.			
Ambiente	Operación normal.			
Estímulo	El usuario hace uso de las funcionalidades del sistema.			
Respuesta	No hay cambios en la implementación de dicha funcionalidad.			
Decisiones arquitectónicas	Riesgo	Punto de Sensibilidad	Punto de Intercambio	No Riesgo
		S1		
Explicación	La integridad de la Información resulta extremadamente importante en los sistemas informáticos. Para mantener la integridad de la información se restringe según el rol y los servicios instalados al usuario y si una entidad solicitante es anónima, el sistema devuelve un error *forbidden* al suscriptor.			

Tabla 2 Escenario #1 Atributo Seguridad

Escenario #: 2	**El sistema mediante el servicio PubSub determina los privilegios a los clientes y aplicación que están autorizados a publicar y suscribirse.**			
Atributo(s)	Funcionalidad-Seguridad.			
Ambiente	Operación normal.			
Estímulo	El usuario hace uso de las funcionalidades del sistema.			
Respuesta	No hay cambios en la implementación de dicha funcionalidad.			
Decisiones arquitectónicas	Riesgo	Punto de Sensibilidad	Punto de Intercambio	No Riesgo
		S2		
Explicación				

Tabla 3 Escenario #2 Atributos Funcionalidad-Seguridad

Escenario #:3	**El sistema debe aceptar los datos introducidos por el usuario solo cuando estos sean correctos.**

Atributo(s)	Funcionalidad-Seguridad			
Ambiente	Operación normal.			
Estímulo	El usuario introduce datos al sistema.			
Respuesta	El sistema acepta los datos insertados correctamente, validando la entrada de los campos de la aplicación.			
Decisiones arquitectónicas	Riesgo	Punto de Sensibilidad	Punto de Intercambio	No Riesgo
		S3		
Explicación	Se lleva a cabo la validación de los formularios para alcanzar elevados niveles de confiabilidad sobre los datos. El sistema debe mostrarle un mensaje de error en el caso que el usuario introduzca los datos incorrectamente, esto le brinda la oportunidad al usuario de corregir dichos datos.			

Tabla 4 Escenario #3 Atributos Funcionalidad-Seguridad

Escenario #: 4	**El sistema cumple con los RF y RNF solicitados por el cliente**			
Atributo(s)	Funcionalidad-Adecuación			
Ambiente	Operación normal.			
Estímulo	El usuario hace uso de las funcionalidades del sistema.			
Respuesta	No hay cambios en la implementación de dicha funcionalidad.			
Decisiones arquitectónicas	Riesgo	Punto de Sensibilidad	Punto de Intercambio	No Riesgo
		S4		
Explicación	La arquitectura propuesta satisface con los RF y RNF plateados por los clientes.			

Tabla 5 Escenario #4 Atributos Funcionalidad-Adecuación

Escenario #:5	**Debe ser capaz de integrarse a otros sistemas, permitiendo la comunicación en tiempo real.**
Atributo(s)	Funcionalidad, Seguridad-Interoperabilidad, Fiabilidad-Tolerancia a fallos, Eficiencia-Usabilidad de Recursos, Mantenibilidad-Estabilidad
Ambiente	Operación normal.
Estímulo	Necesidad de interactuar: brindar o recibir información con otros sistemas desplegados en el componente.

Respuesta	Las decisiones arquitectónicas tomadas			
Decisiones arquitectónicas	Riesgo	Punto de Sensibilidad	Punto de Intercambio	No Riesgo
	R1		**T1**	
Explicación	El sistema se concibió con la idea de que en un futuro este pueda ser capaz de intercambiar flujo de datos con otros sistemas desplegados en el componente y a la hora de tomar las decisiones arquitectónicas se pensó en dicho detalle.			

Tabla 6 Escenario #5 Atributos Funcionalidad, Seguridad-Interoperabilidad, Fiabilidad-Tolerancia a fallos, Eficiencia-Usabilidad de Recursos, Mantenibilidad-Estabilidad

Escenario #: 6	**Al ocurrir un error en el servicio PubSub, la operación queda cancelada por lo que se le envía un mensaje al usuario y el sistema deberá retornar a su estado inicial antes de iniciada la petición.**			
Atributo(s)	Fiabilidad- Tolerancia a fallos – Recuperación			
Ambiente	Operación normal.			
Estímulo	Se produce un error durante la publicación-suscripción.			
Respuesta	El sistema debe ser capaz volver al estado inicial antes de realizada la petición.			
Decisiones arquitectónicas	Riesgo	Punto de Sensibilidad	Punto de Intercambio	No Riesgo
			T2	
Explicación	Lograr que el sistema retorne a su estado inicial tras la ocurrencia de algún error fue uno de los puntos en los que se tuvieron en cuenta a la hora de tomar decisiones arquitectónicas en este caso a la hora de definir los RNF, producto que influye en atributos claves como la seguridad, funcionalidad y fiabilidad.			

Tabla 7 Escenario #6 Atributos Fiabilidad- Tolerancia a fallos - Recuperación

Escenario #:7	**El sistema debe tener la capacidad de ser atractivo, entendible para el usuario.**			
Atributo(s)	Usabilidad-Comprensibilidad-Aprendibilidad-Atractividad			
Ambiente	Operación normal.			
Estímulo	El usuario hace uso de las funcionalidades del sistema.			
Respuesta	No hay cambios en la implementación de dicha funcionalidad.			
Decisiones	Riesgo	Punto de Sensibilidad	Punto de Intercambio	No Riesgo

arquitectónicas				
			T3	
Explicación	La utilización del frameworks Dojo permitirá obtener aplicaciones entendibles y atractivas para el usuario.			

Tabla 8 Escenario #7 Atributos Usabilidad-Comprensibilidad-Aprendibilidad-Atractividad

Escenario #: 8	**El sistema debe ser capaz de actualizar bidireccionalmente la información manejada durante el período en que no se mantuvo la conexión.**			
Atributo(s)	Eficiencia-Usabilidad de Recursos, Funcionalidad-Adecuación.			
Ambiente	Operación normal.			
Estímulo	Un período determinado de tiempo sin conexión			
Respuesta	No hay cambios en la implementación de dicha funcionalidad.			
Decisiones arquitectónicas	Riesgo	Punto de Sensibilidad	Punto de Intercambio	No Riesgo
	R2		**T4**	
Explicación	El sistema debe ser capaz de seguir brindando prestaciones al usuario aunque este haya perdido la conexión, una vez restablecido todo y vuelto a la normalidad la aplicación debe permitir realizar la actualización de toda la información manejadas durante ese período de tiempo. Los servicios basados en el protocolo BOSH implementan estas características.			

Tabla 9 Escenario #8 Atributos Eficiencia-Usabilidad de Recursos

Escenario #:9	**Los tiempos de respuesta a las peticiones realizadas por los usuarios deben ser en tiempo real.**			
Atributo(s)	Eficiencia, Rendimiento-Tiempo de Respuesta.			
Ambiente	Operación normal.			
Estímulo	El usuario necesita visualizar los datos en tiempo real.			
Respuesta	No hay cambios en la implementación de dicha funcionalidad.			
Decisiones arquitectónicas	Riesgo	Punto de Sensibilidad	Punto de Intercambio	No Riesgo
			T5	
Explicación	El sistema brinda sus funcionalidades a un número considerable de usuarios y debe hacerlo con un elevado rendimiento y optimización. El			

	servidor Ejabberd (XMPP) es ligero. El mecanismo de *Long Polling* que emplea BOSH facilita eficiencia y mínima latencia para transportar los datos bidireccionalmente.

Tabla 10 Escenario #9 Atributos Eficiencia, Rendimiento-Tiempo de Respuesta

Escenario #: 10	**El sistema debe permitir que se le puedan adicionar componentes, módulos o nuevas funcionalidades.**			
Atributo(s)	Mantenibilidad-Cambiabilidad, Configurabilidad.			
Ambiente	Operación normal.			
Estímulo	Necesidad de adicionar funcionalidades al sistema.			
Respuesta	La arquitectura definida soporta la incorporación de nuevas funcionalidades, módulos al sistema.			
Decisiones arquitectónicas	Riesgo	Punto de Sensibilidad	Punto de Intercambio	No Riesgo
		S5		
Explicación	La arquitectura diseña es capaz de soportar en ambos sistemas la posibilidad de adicionarle otros módulos e incluso funcionalidades a los módulos existentes. 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.			

Tabla 11 Escenario #10 Atributos Mantenibilidad-Cambiabilidad, Configurabilidad

Escenario #: 11	**Los componentes pueden instalarse fácilmente en todos los ambientes donde debe funcionar, o sea Linux, por su característica de ser código abierto.**			
Atributo(s)	Mantenibilidad-Facilidad de adaptación al cambio, Portabilidad			
Ambiente	Operación normal.			
Estímulo	Despliegue de las aplicaciones.			
Respuesta	La arquitectura definida soporta el despliegue en los sistemas operativos Linux. Se escogió como lenguaje de programación PHP y el servidor Ejabberd para manejar los datos.			
Decisiones arquitectónicas	Riesgo	Punto de Sensibilidad	Punto de Intercambio	No Riesgo

		S6		
Explicación	Las decisiones arquitectónicas tomadas influyen en la portabilidad de un producto de software, los elementos de la plataforma tecnológica se definieron pensando en la posibilidad de un cambio de sistema operativo.			

Tabla 12 Escenario #11 Atributos Mantenibilidad-Facilidad de adaptación al cambio, Portabilidad

Escenario #: 12	**Es fácil instalar las actualizaciones del sistema**			
Atributo(s)	Portabilidad			
Ambiente	Operación normal.			
Estímulo	Necesidad de realizar actualizaciones al sistema.			
Respuesta	El sistema una vez desplegado permite realizar las actualizaciones pertinentes.			
Decisiones arquitectónicas	Riesgo	Punto de Sensibilidad	Punto de Intercambio	No Riesgo
		S7		
Explicación				

Tabla 13 Escenario #12 Atributos Portabilidad

3.3 Toma de decisiones

Se identificaron 2 riesgos en las tablas de los escenarios 5 y 8. Una vez identificados los riesgos presentes en la arquitectura resulta necesario efectuar la toma de decisiones por parte de los *stakeholders* y el equipo que intervino en el proceso de ejecución de las pruebas de concepto de la arquitectura: arquitecto de software, líder de proyecto y algunos desarrolladores.

La toma de decisiones se ejecuta con el objetivo de mitigar los riesgos en la arquitectura desde el punto de vista de los atributos de calidad analizados y dar paso a la construcción del sistema. Como principales decisiones a tomar se aprobaron: definir políticas para la interacción del sistema con otras aplicaciones, así como para el proceso de actualización bidireccional con el objetivo de mantener la seguridad e integridad de los datos que se manejan; desarrollar un proceso de auditoría y control al diseño arquitectónico del sistema, así como analizar las consecuencias que pueden arrojar cambios significativos en las decisiones arquitectónicas definidas.

Conclusiones parciales del capítulo

En el presente capítulo se han analizado los principales elementos relacionados con la evaluación de la arquitectura de software diseñada, propiciando una correcta y adecuada selección de seis atributos

de calidad definidos por la ISO/IEC 9126: funcionalidad, fiabilidad, usabilidad, eficiencia, mantenibilidad y portabilidad; todos relacionados estrechamente con la arquitectura de software.

Se aplicó el método de evaluación de la arquitectura ATAM como parte del proceso de evaluación temprana, además de aplicar la técnica basada en escenarios y el procedimiento del árbol de utilidad. De un total de 12 escenarios clasificados con una prioridad entre alta, media y baja, el método arrojó finalmente 7 puntos de sensibilidad, 2 puntos de riesgos, 5 puntos de desventajas o *tradeoff*, por lo que se decidió a dar paso a la implementación del sistema. Se arribó a la conclusión de que el sistema es principalmente funcional.

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 arquitectura de software permite organizar el proceso de desarrollo y definir la estructura que tendrá el sistema.

Como resultado se obtuvo el diseño de la arquitectura del Componente de Comunicación de Datos sobre la Web en tiempo real. Se utilizó el estilo de arquitectura en Capas y *Peer to Peer*. 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.

Como parte de la evaluación de la arquitectura diseñada se aplicó el método de evaluación de la arquitectura ATAM, el cual contribuyó a que se alcanzara un sistema funcional y se diera paso a la implementación del sistema. Con la evaluación de la arquitectura quedó evidenciada la forma en que la arquitectura propuesta hizo tratamiento de los diferentes atributos de calidad analizados.

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 del grupo de arquitectura de la facultad y de la universidad la generalización de la arquitectura propuesta para el desarrollo de nuevas aplicaciones informáticas con características similares.

REFERENCIAS BIBLIOGRÁFICAS

1. Kaplan, R.y.N., David, *Balanced Scorecard: Measures that Drive Performance*. 1992: Boston.
2. Alumnos, I.-R.d.A. (marzo de 2001) *Pasado, presente y futuro del Cuadro de Mando Integral*.
3. Leonard, L.B.C.A. *El Cuadro de Mando Integral una necesidad en las empresas cubanas*.
4. Kruchten, P., *The "4+1" View Model of Software Architecture*. 1995.
5. Len Bass, P.C., Rick Kazman, *Software Architecture in Practice*. Second Edition ed. 2003: Addison-Wesley Professional.
6. 1471-2000, I., *Recommended practice for architectural description of software-intensive systems*. 2000, IEEE.
7. Ivar Jacobson, G.B., James Rumbaugh, *The Unified Software Development Process*. 1999: Addison-Wesley Professional.
8. Garlan, D., *Software Architecture: A Roadmap*. The future of software engineering, ed. A. Finkelstein. 2000: ACM Press.
9. Carlos Reynoso, N.K. *Estilos y Patrones en la Estrategia de Arquitectura de Microsoft*. 2004; Available from: http://www.willydev.net/descargas/prev/Estiloypatron.pdf.
10. Robert Allen, D.G., *The Wright Architectural Description Language*. Verano de 1996, Carnegie Mellon University.
11. Mark Klein, R.K., *Attribute-based architectural styles*. Octubre de 1999, Carnegie Mellon University.
12. César de la Torre, U.Z.C., Miguel A. Ramos, Javier C. Nelson, *GUÍA DE ARQUITECTURA N-CAPAS ORIENTADA AL DOMINIO CON .NET*. Krasis ed. 2010. 433.
13. Sarver, T. *Pattern Refactoring Workshop*. 2000; Available from: http://www.laputan.org/patterns/positions/Sarver.html.
14. Welicki, L.E., *Meta-Especificación y Catalogación de Patrones de Software con Lenguajes de Dominio Específico y Modelos de Objetos Adaptativos: Una Vía para la Gestión del Conocimiento en la Ingeniería del Software*, in *Facultad de Informática*. 2006, Universidad Pontificia de Salamanca: Madrid. p. 449.
15. Patencier, F.a.Z., Francois, *The Definitive Guide to Symfony*, ed. Apress. 2007.
16. David Garlan, M.S. (January 1994) *An Introduction to Software Architecture*. CMU Software Engineering Institute Technical Report CMU/SEI-94-TR-21, ESC-TR-94-21.
17. Larman, C., *UML y PATRONES* 1999, México.

18. Erich Gamma, R.H., Ralph Johnson, John Vlissides, *Design Patterns: Elements of reusable object-oriented software*. 1995.
19. Alberto José, J.T. *Uso de XMPP para el transporte de información cooperativa y de contexto*. 2009; Available from: http://upcommons.upc.edu/pfc/bitstream/2099.1/6907/1/memoria.pdf.
20. Robles, L.A. *Sistema Seguidor de Objetos*. 2003; Available from: http://ccc.inaoep.mx/~labvision/doo/proy/T72.pdf.
21. UML, O.M.G.-. 2009; Available from: http://www.uml.org/.
22. Sánchez, D.M. *Técnicas y Metodologías en el desarrollo de Software*. 2007; Available from: http://kybele.escet.urjc.es/documentos/HC/HC4GL2007-T2-TecnicasI.pdf.
23. *Eclipse Process Framework (EPF)*. Available from: http://epf.eclipse.org.
24. *Rational Rose 2000e Using Rose*. 2000:[Available from: http://www.vico.org/aRecursos/Rational/Rose_usingrose.pdf.
25. Salazar, J.S. *Curso de Java*. 2005; Available from: http://tikal.cifn.unam.mx/%7Ejsegura/LCGII/java3.htm.
26. PHP. *News Archive - 2007*. 2007; Available from: http://www.php.net/archive/2007.php.
27. García, L. *Sistema de control de vesiones: SUBVERSION*. enero 2008; Available from: http://observatorio.cnice.mec.es/modules.php?op=modload&name=New&file=article&sid=548.
28. *Ext - A foundation you can build on*. Available from: http://www.extjs.com/5064.
29. NetBeans. Available from: http://netbeans.org.
30. *RFC 3920: Extensible Messaging and Presence Protocol (XMPP)*. Available from: http://tools.ietf.org/html/rfc3921.
31. *RFC 3921: Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence*. Available from: http://tools.ietf.org/html/rfc3921.
32. Peter Saint-Andre, K.S., and Remko Tronçon, *XMPP: The Definitive Guide*. First ed. April 2009: O'Reilly Media.
33. Ian Paterson, P.S.-A. *XEP-0206: XMPP Over BOSH*. [Standards Track]; Available from: http://xmpp.org/extensions/xep-0206.html.
34. Moffitt, J.; Available from: http://strophe.im/strophejs/.
35. Gómez, O.S.G., *Evaluando Arquitecturas de Software*. Vol. Parte 1. Panorama General. 2007, México: Brainworx S.A.
36. Gustavo Andrés Brey, G.E., Nicolas Passerini y Juan Arias, *Arquitectura de Proyectos de IT. Evaluación de Arquitecturas*, in *Departamento de Sistemas*. 2005, Universidad Tecnológica Nacional. Facultad Regional de Buenos Aires: Buenos Aires.
37. Erika Camacho, F.C., Gabriel Nuñez, *Arquitecturas de Software*. 2004.

38. Kazman R., C.P., Klein M., *Evaluating Software Architectures. Methods and case studies*. 2001: Addison Wesley.
39. Bosch, *Design & Use of Software Architectures*. 2003: Addison-Wesley.
40. Kazman Rick, K.M.a.C.P., *ATAM: Method for Architecture Evaluation*. 2000, Carnegie Mellon, Software Engineering Institute.
41. Clement, P. *SEI (Software Engeniering Institute)* 2000; Available from: http://www.sei.cmu.edu/publications/documents/00.reports/00tn009.html.
42. Reynoso, C.B. *Introducción a la Arquitectura de Software*. Marzo de 2004; Available from: http://www.willydev.net/descargas/prev/IntroArq.pdf.
43. Alarcón, R., *Diseño Orientado a Objetos con UML*. 2000, Madrid.
44. 9126-1:2001, I.I. *Ingeniería de software - La calidad del producto - Parte 1: Modelo de Calidad*. Available from: http://www.iso.org.

BIBLIOGRAFÍA

(1). Álvarez Medina, M. T., & Moreno Velarde, S. A. EL BALANCED SCORECARD, UNA HERRAMIENTA PARA LA PLANEACIÓN ESTRATATEGICA.

(2). Clements, P., Kazman, R., & Klein, M. (2002). Evaluating Software Architectures. Adisson Wesley.

(3). Davila, A. (1999). El Cuadro de Mando Integral. Revista Antiguos Alumnos IESE.

(4). Jacobson, I., Booch, G., & Rumbaugh, J. (2004). El Proceso Unificado de Desarrollo de Software. La Habana: Félix Varela.

(5). Martin, R. C. Design Principles and Design Patterns.

(6). Meier, J. D. (2008). Application Architecture Guide 2.0. Microsoft Press.

(7). Moffitt, J. (2010). Professional XMPP Programming with JavaScript and jQuery. Indianapolis, Indiana: Wiley Publishing.

(8). NetBeans. (2009). Recuperado el 12 de Febrero de 2011, de www.netbeans.com

(9). Pressman, R. Ingeniería del Software. Un enfoque práctico. La Habana: Félix Varela.

(10). Saint-Andre, P., & Tronçon, R. (2009). Building Real-Time Applications whit Jabber Technologies. O Relly.

(11). Sommerville, I. (2004). Ingeniería del Software. España: Addison-Wesley.

(12). Trejo Vargas, J. M., & Icedo Ojeda, R. V. (2003). Software Architecture Assesment.