Universidad de las Ciencias Informáticas
Facultad 6

Título:

“Definición de la arquitectura de software del Grupo de Desarrollo para la Gestión de Equipos Médicos.”

Trabajo de Diploma para optar por el título de

Ingeniero en Ciencias Informáticas

Autor:
Alberto Mendoza Garnache

Tutores:
Ing. Dennys Hernández Peña.
Ing. Arieskien Mendoza Guerra.
Ing. Yanelis Ramírez Hernández.

Co-Tutor:
Ing. Adonis Ricardo Rosales García

Ciudad de la Habana, Junio de 2009
“Año del 50 aniversario del triunfo de la Revolución”

"La arquitectura es el arte de organizar el espacio. Como meta debe proponer la creación de relaciones nuevas entre el hombre, el espacio y la técnica"

Hans Scharoun

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 _____.

Alberto Mendoza Garnache
_____
[Firma autor]

Ing. Dennys J. Hernández Peña
_____
[Firma tutor]

Ing. Yanelis Ramírez Hernández
_____
[Firma tutor]

Ing. Arieskien Mendoza Guerra
_____
[Firma tutor]

Ing. Adonis Ricardo Rosales García
_____
[Firma co-tutor]

DATOS DE CONTACTO

Nombre: Alberto

Apellidos: Mendoza Garnache

Correo: agarnache@estudiantes.uci.cu

Nombre: Yanelis

Apellidos: Ramírez Hernández

Título universitario: Ingeniero en Ciencias Informáticas

Año de graduado: 2008

Correo: yramirezh@uci.cu

Nombre: Dennys

Apellidos: Hernández Peña

Título universitario: Ingeniero Informático

Año de graduado: 2005

Correo: dhernandezp@uci.cu

Nombre: Arieskien

Apellidos: Mendoza Guerra

Título universitario: Ingeniero en Ciencias Informáticas

Año de graduado: 2008

Correo: amendoza@uci.cu

Nombre: Adonis Ricardo

Apellidos: Rosales García

Título universitario: Ingeniero en Ciencias Informáticas

Año de graduado: 2008

Correo: arrosales@uci.cu

AGRADECIMIENTOS

A Dios por su infinito amor y porque sin él no hubiese llegado.

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 mi familia por su confianza y apoyo constante.

A mi tía Amada por ser mi segunda madre en la vida.

A las personas que me han ayudado a salir adelante en los momentos más difíciles.

A mis tutores por su entrega y paciencia para conmigo.

A mis amigos que estuvieron a mi lado en los buenos y menos buenos momentos: Diana, Linet, Yislén, Maidelys, Yanet, Aleida, Lily, Mily, Yisel, Ive, Veylys, Lourdes, Liuber, Peter, Rey, Rene...

A todos mis compañeros y hermanos del proyecto GDEM por llegar a formar la familia que somos hoy.

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

DEDICATORIA

A mi abuela Aida quien llevo en mi corazón por estar siempre a mi lado y ser mi inspiración.

A mi madre y mi hermano por ser la fuente de vida y motivo de mí existir.

A mi familia que siempre esperó lo mejor de mí.

RESUMEN

La investigación surge en el marco de trabajo del proyecto Grupo de Desarrollo para la Gestión de Equipos Médicos (GDEM) producto de las necesidades del grupo en materia de estandarización e integración de las aplicaciones para obtener componentes altamente integrados y proveer soluciones escalables y confiables.

La arquitectura definida provee los elementos necesarios para el desarrollo de software dentro de GDEM. Su éxito viene dado por las decisiones arquitectónicas que fueron tomadas a lo largo de ciclo de vida del desarrollo del producto, por la definición de la plataforma tecnológica, de los lineamientos y mecanismos arquitectónicos, así como la descripción de la arquitectura por medio de las vistas de la arquitectura.

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.

Se realizó una evaluación de la arquitectura definida aplicando el método de evaluación de la arquitectura Architecture Trade-off Analisys Method (ATAM) conjuntamente con el procedimiento Árbol de Utilidad y la técnica Basada en Escenarios obteniendo resultados satisfactorios.

El presente trabajo constituye una guía para los desarrolladores que deseen emplear la combinación del framework de desarrollo Symfony con el framework de presentación ExtJS.

PALABRAS CLAVES

Arquitectura de software, plataforma tecnológica, entorno de desarrollo, framework.

ÍNDICE

INTRODUCCIÓN

En los primeros años de la producción de software no existía el diseño del sistema como una etapa independiente de la programación, pensar en la descomposición del sistema antes de programar se comienza a proponer, investigar y aplicar a principios de la década del 70 cuando se empieza a distinguir entre los pequeños y grandes sistemas. Precisamente uno de los pioneros en esta área fue David L. Parnas con sus trabajos seminales en ocultación de información y desarrollo de familias de programas, seguido por el trabajo de Liskov y Guttag referido al diseño basado en tipos abstractos de datos. En la misma época pero con un impacto en la industria mucho mayor Tom DeMarco, Edward Yourdon, Larry Constantine y Glenford Myers desarrollaron y consolidaron el análisis y diseño estructurado. En este proceso de evolución mucho se intentó por mejorar aplicando prácticas por ejemplo:

- La **técnica de codificar y corregir**, donde primero se implementaba algo y luego se pensaba a cerca de los requisitos, diseño, validación y mantenimiento.
- El **desarrollo en espiral**, donde se toma en consideración explícitamente el riesgo, actividad muy importante en la administración del proyecto.
- El **modelo en cascada**, el toma especificación, desarrollo, validación y evolución y las representa como fases separadas del proceso.
- El **desarrollo evolutivo**, el cual propone la implantación del sistema inicial, exponerla al usuario y refinarla en n versiones.
- También el **desarrollo basado en la reutilización** el cual pospone decisiones en los requisitos hasta adquirir experiencia, combinando así el modelo cascada y el evolutivo.

Teniendo todas estas prácticas como factor común: en la especificación de software, se debe definir la funcionalidad y restricciones operacionales que debe cumplir el sistema. El diseño e implementación, donde se diseña y construye el software de acuerdo a la especificación. La validación, donde el software debe validarse, para asegurar que cumpla con lo que quiere el cliente. Finalmente la evolución, donde el software debe evolucionar, para adaptarse a las necesidades del cliente.

En la década del 90 la comunidad de profesionales relacionados con la producción de software se comenzó a plantear la necesidad de expresar la estructura del sistema en un nivel de abstracción superior, algunos investigadores, principalmente ligados a la Carnegie-Mellon University y al Software Engineering Institute (SEI), comienzan a ver la necesidad de investigar y desarrollar un nivel de abstracción superior al del diseño, al que llamaron **arquitectura de software**.

El estado actual de la ingeniería de software está dado por un desarrollo lento, costoso, con un aumento en la demanda y magnitud de las solicitudes. La arquitectura de software en la actualidad es un conjunto inmenso y heterogéneo de áreas de investigación teórica y de formulación práctica. Tiende a redefinir todos y cada uno de los aspectos de la disciplina madre, la ingeniería de software, sólo que a un mayor nivel de abstracción y agregando una nueva dimensión reflexiva en lo que concierne a la fundamentación formal del proceso. Ahora que la arquitectura de software se ha abismado en el desarrollo de metodologías, hace falta, por ejemplo, establecer con más claridad en qué difieren sus elaboraciones en torno al análisis de requerimientos y al diseño. En este sentido, se espera una lista sistemática y exhaustiva que describa los dominios de responsabilidad de la disciplina, así como un examen del riesgo de duplicación de esfuerzos entre campos disciplinarios mal comunicados, una situación que a primera vista parecería contradictoria con el principio de reusabilidad.

La arquitectura en los proyectos de desarrollo de software, constituye en la actualidad un tema extremadamente polémico; una incorrecta e inadecuada definición de la arquitectura de software puede llevar al caos a un proyecto cualquiera. La Universidad de las Ciencias Informática está llamada a convertirse en una potencia informática de desarrolladores de software. En este sentido se trabaja por lograr que el trabajo en los proyectos productivos sea satisfactorio, obtener productos con la calidad requerida y que cumplan con las expectativas de los clientes que solicitan el software.

Actualmente en la universidad se desarrollan varios proyectos tanto de corte nacional como de exportación, uno de ellos es el Grupo de Desarrollo para la Gestión de Equipos Médicos de la facultad 6, propio del polo científico de gestión de información biomédica, en colaboración con centros pertenecientes al Ministerio de Salud Pública, el Centro de Control Estatal de Equipos Médicos y el Centro de Ingeniería Clínica y Electromedicina; cuyo propósito es mejorar la calidad de vida de la población cubana.

Para el CCEEM, el grupo se propuso asegurar la continuidad del proceso estatal del equipamiento médico a través del perfeccionamiento del sistema y vigilancia, considerando los niveles estructurales y organizativos del Sistema Nacional de Salud Pública (SNS), así como la creación de redes de comunicación nacional e internacional. Estructurar por componentes, respondiendo a las tendencias arquitectónicas actuales y desarrollado con herramientas y tecnologías de software libre, sin violar las normas establecidas por el SNS.

El CCEEM contempla dentro de sus objetivos fundamentales y de acuerdo al nivel de la práctica internacional la implementación de un Sistema de Vigilancia para los Equipos Médicos como parte del seguimiento postmercado que permita identificar y alertar sobre eventos adversos ocurridos con equipos médicos dentro del territorio nacional. Su implementación se basa en un Sistema de Información Nacional de Vigilancia que sistematiza, colecta datos, procesa y difunde las informaciones sobre incidentes, fallas, "problemas", o eventos adversos asociados al equipamiento médico.

Para ello se ha desarrollado una aplicación informática para el control y seguimiento de los equipos médicos instalados en el Sistema Nacional de Salud (SNS) que permiten controlar la gestión de la información de estos equipos en la etapa de post comercialización. El sistema está compuesto actualmente por seis módulos de aplicación entre los cuales se encuentra un módulo para la gestión de la información del departamento de supervisión, un módulo para la inscripción de fabricantes y registro de equipos médicos, el módulo de información de equipos médicos de radiofísica, un módulo para el reporte de eventos adversos por parte de las instituciones de salud y otro por parte del fabricante de equipos médicos, así como un generador de encuestas para el monitoreo de los equipos médicos en el SNS.

Dicho sistema se desarrollará siguiendo las pautas propuestas por el SNS para su informatización. Con la culminación y puesta en práctica del mismo se habrá dado un paso de avance para la automatización y control de la Vigilancia de Equipos Médicos por el CCEEM en el SNS.

Para el CICEM, se pretende desarrollar el Sistema de Gestión para Ingeniería Clínica y Electromedicina, aplicación informática para el uso del Sistema Nacional de Electromedicina. Sus prestaciones van desde el aseguramiento técnico, inventario, gestión de estadística técnica, gestión de Recursos Humanos (RRHH) hasta la gestión logística de piezas de repuesto. Aportando un amplia gama de funcionalidades aseguradoras del proceso de gestión de la tecnología médica. Proveer una solución eficaz para alcanzar elevados niveles de eficiencia en los procesos de gestión de la actividad técnica, abarcando los principales conceptos exigidos internacionalmente para esta actividad entre los que se destacan: organización, planificación, control y análisis. Su desarrollo responde a las tendencias arquitectónicas de actualidad con la utilización de herramientas y tecnologías de software libre, cumpliendo con las normas establecidas por el SNS.

Se tiene como meta suprema conformar un Sistema Integral de Gestión para la Vigilancia de Equipos Médicos (SIGVEM), así como proveer soluciones simples, escalables y confiables. Para alcanzar las metas trazadas resulta necesario lograr en el grupo una estandarización e integrar cada uno de los módulos de los dos proyectos existentes actualmente, coordinar la integración de las aplicaciones para lograr que se pueda reutilizar código, que se obtengan componentes altamente integrados, evitar que se realicen acciones innecesarias que provoquen retrasos en los proyectos, que abarquen más allá del alcance del negocio y contribuir a la gestión de la configuración, todo esto a través de la definición de una arquitectura sólida, robusta y bien concebida. La arquitectura, las normas, los lineamientos arquitectónicos y las tecnologías, entre otros aspectos relevantes para el desarrollo de aplicaciones informáticas para el SNS definidos por el grupo de arquitectura conformado por la dirección informática del MINSAP, Ministerio de la Informática y las Comunicaciones (MIC), INFOMED, SOFTEL y la UCI no se puede adaptar a las necesidades planteadas, por lo que se tiene que incorporar elementos a la plataforma tecnológica para el desarrollo, la cual debe ser consultada y evaluada por este grupo de arquitectura teniendo en cuenta el impacto directo que pueda tener sobre la plataforma de despliegue.

La situación anteriormente descrita atenta contra el mejor desempeño de las tareas que realizan estas importantes instituciones y refleja la necesidad de estudiar una posible solución a la misma. La realización del presente trabajo va encaminada a mejorar la problemática relacionada con la definición de una arquitectura estable, por lo que se plantea como **problema científico**: ¿Cómo definir la arquitectura de software del Grupo de Desarrollo para la Gestión de Equipos Médicos?

La investigación tiene como **objeto de estudio** la arquitectura de software, enmarcado en el **campo de acción** la arquitectura de software para aplicaciones de salud para la gestión de equipos médicos.

El **objetivo general** de este trabajo es: definir la arquitectura de software para la línea de productos de salud para la gestión de equipos médicos.

En correspondencia con ello, se plantean como objetivos específicos:

- Definir la línea base de la arquitectura.
- Describir la arquitectura de software del Grupo de Desarrollo para la Gestión de Equipos Médicos.
- Evaluar la arquitectura de software diseñada para el Grupo de Desarrollo para la Gestión de Equipos Médicos.

Para el cumplimiento de estos objetivos se desglosaron las siguientes **tareas**:

- Estudio y selección del estilo y de los patrones.
- Combinación de patrones.
- Fundamentación de la utilización y aplicación de los patrones.
- Definición de las herramientas y tecnologías a utilizar para el desarrollo.
- Diseño de la propuesta arquitectónica que cumpla con los requerimientos de la aplicación y que garantice el desarrollo paralelo y la reutilización de componentes de la misma.
- Diseño de las vistas del sistema.
- Representación del diseño arquitectónico.
- Definición de las estrategias de comunicación e integración de los módulos del proyecto.
- Estudio y selección de los posibles métodos basados en la arquitectura.
- Aplicación de los métodos seleccionados para evaluar la arquitectura de software definida.

El presente documento se divide en varios capítulos:

- El Capítulo 1 con la fundamentación teórica, donde se analizan los principales conceptos asociados al objeto de estudio: arquitectura de software, así como lo referente a los estilos arquitectónicos, patrones, lenguaje de descripción de arquitectura.

- El Capítulo 2 arquitectura del sistema, se dedica a la plataforma de desarrollo, plataforma de despliegue, requerimientos de hardware, estructura y organización del proyecto. Así como a representar la descripción de la arquitectura de software a través de la vistas de la arquitectura.

- El Capítulo 3 evaluación de la arquitectura, muestra un estudio de la factibilidad de la arquitectura definida. Se expone lo referente a los principales métodos basados en la arquitectura utilizados para evaluar las arquitecturas de los proyectos a nivel mundial. Se describe además el método seleccionado para aplicarlo en el proyecto, logrando así la validación de la arquitectura de software definida.

CAPÍTULO 1 FUNDAMENTACIÓN TEÓRICA

Introducción

En el presente capítulo se plantean los principales conceptos relacionados con el objeto de estudio y que resultan vitales para entender posteriormente la solución propuesta. Se abordará sobre el análisis del estado del arte de la arquitectura de software en lo adelante, partiendo del estudio sus principales concepciones, directrices de los estilos arquitectónicos, los patrones así como los lenguajes de descripción de la arquitectura.

1.1 Arquitectura de software

La arquitectura asocia las capacidades del sistema especificadas en el requerimiento con los componentes del sistema que habrán de implementarla. La descripción arquitectónica incluye componentes y conectores (en términos de estilos) y la definición de operadores que crean sistemas a partir de subsistemas o, en otros términos, componen estilos complejos a partir de estilos simples.

La arquitectura, conformada por diferentes visiones del sistema, constituye un modelo de cómo está estructurado dicho sistema, sirviendo de comunicación entre las personas involucradas en el desarrollo y ayudando a realizar diversos análisis que orienten el proceso de toma de decisiones. Para que la arquitectura se convierta en una herramienta útil dentro del desarrollo y mantenimiento de los sistemas de software es necesario que se cuente con una manera precisa de representarla.

La arquitectura de software conjuntamente con los patrones y los métodos ortodoxos conforman la triada de grandes temas de la ingeniería de software. Uno de los progresos más importantes dentro de la producción de software ha sido en los últimos años el desarrollo de la arquitectura de software, la cual permite representar la estructura de un sistema a un nivel mayor que el dado por la programación e incluso por el diseño. Constituye a grandes rasgos, una vista del sistema que incluye los principales componentes del mismo, la conducta de dichos componentes según se la percibe desde el resto del sistema y las formas en que los componentes interactúan y se coordinan para alcanzar la misión del sistema. En un programa o sistema de cómputo, es la estructura o estructuras del sistema que comprenden elementos de software, las propiedades visibles externamente de esos elementos y las relaciones entre ellos. El nivel conceptual más alto de un sistema en su ambiente.

CAPÍTULO 1

FUNDAMENTACIÓN TEÓRICA

A continuación se representan los principales conceptos de arquitectura de software estudiados y que sirven de base para el desarrollo del trabajo:

La arquitectura de software consiste en la estructura o sistema de estructuras, que comprenden los elementos de software, las propiedades externas visibles de esos elementos y la relación entre ellos. (1)

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

La arquitectura de software constituye un puente entre el requerimiento y el código, ocupando el lugar que en los gráficos antiguos se reservaba para el diseño. (3)

"Arquitectura es la estructura de los componentes más significativos de un sistema interactuando a través de interfaces con otros componentes conformados por componentes sucesivamente pequeños e interfaces". [RUP]

"Estructura de los componentes de un programa o sistema, sus interrelaciones, y los principios y reglas que gobiernan su diseño y evolución en el tiempo" (Garlan y Perry, 1995)

"Estructura o estructuras de un sistema, lo que incluye sus componentes de software, las propiedades observables de dichos componentes y las relaciones entre ellos" (Bass, Clements y Kazman, 1998)

La definición oficial de arquitectura del software es la IEEE Std 1471-2000 que reza así: *"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".* (4)

Un componente constituye una parte de una arquitectura de software claramente identificable, independiente a la aplicación en la que se utiliza y de otros componentes, que describe y realiza funciones específicas y claras dentro del contexto de la arquitectura, puede ser modificado durante el diseño, posee una documentación clara que permite conocer sus características, atributos y comportamiento, reutilizable y su interoperabilidad con otros componentes no reduce el nivel de eficiencia de la arquitectura. (5)

Características de los componentes

Unidades de software orientadas a la composición.

- → Incluso en tiempo de ejecución.
- → Funcionalidad y complejidad significativas.
- → Binarios: no son especificaciones ni código fuente.

Integrados en un determinado modelo/plataforma.

- → No tienen sentido de forma aislada.

Implementan una serie de interfaces.

- → Indican la funcionalidad que proporcionan también la que precisan.
- → Permiten la reemplazabilidad.

Reutilizables a partir de su especificación.

- → En contextos distintos a los originales.
- → De manera no previsible para sus constructores.

La arquitectura del software aporta una visión abstracta de alto nivel, posponiendo el detalle de cada uno de los módulos definidos a pasos posteriores del diseño. Es una vista estructural de alto nivel; define un estilo o combinación de estilos para una solución; centra su atención en los requerimientos no funcionales, ya que los requerimientos funcionales se satisfacen mediante el modelado y diseño de la aplicación. Muy relacionada al trabajo en equipo y la correcta planificación y costo del software. Es esencial para el éxito o fracaso de un proyecto.

CAPÍTULO 1

FUNDAMENTACIÓN TEÓRICA

Como objeto de estudio, la arquitectura software constituye una subdisciplina de la ingeniería del software. Pasa por todos los flujos de trabajo, se va perfeccionando, refinando a medida que se vayan realizando tareas particulares a lo largo del ciclo de vida del software; es importante porque representa distintas vistas que inciden en la implementación futura del sistema. Es el esqueleto o base de una aplicación, en la que se analiza la aplicación desde varios puntos de vista. Tiene como objetivo reutilizar tantos los componentes como los patrones estructurales. En otras palabras podría ser: “Las habilidades de seleccionar, combinar e integrar las tecnologías informáticas existentes para construir un software teniendo en cuenta los requisitos funcionales y no funcionales del cliente”.

La arquitectura no está limitada a la estructura interna, sino que tiene en cuenta el sistema como un todo en el contexto de los usuarios y el desarrollo. Constituye un modelo relativamente pequeño e intelectualmente comprensible de cómo trabajan juntos sus componentes, de ahí que es capaz de crujir un conjunto de necesidades tales como:

- Permite comprender el sistema en su totalidad,
- Organizar el desarrollo, dividirlos en partes más manejables y organizar el equipo que lo va a desarrollar. La división del sistema en subsistemas con interfaces claramente definidas con un responsable establecido para cada subsistema reduce la carga de comunicación entre los grupos de trabajo, por tanto, las interfaces permiten el progreso independiente de software a ambos lados de las mismas.
- Fomenta la reutilización, o sea, es la clave para el reuso: transferible, abstracción reusable. Hace evolucionar el sistema en la medida que se hayan las nuevas versiones.
- Hacer evolucionar el sistema, Proporciona la seguridad de que el sistema evolucionará ante modificaciones de diseño o implementación o ante la introducción de nuevas funcionalidades.

Perspectivas de la arquitectura

Negocio: Describe el funcionamiento interno del negocio central de la organización.

Aplicación: Muestra las aplicaciones de la organización, su funcionalidad y relaciones.

Información: Describe la información que maneja la organización.

Tecnología: Describe la estructura de hardware y software de base que da soporte informático a la organización.

La arquitectura de un software puede concebirse como aquella estructura del programa que cohesiona las funcionalidades más críticas y relevantes (necesarias para el sistema), y que sirve de soporte al resto de funcionalidades finales (necesarias para el usuario). Su especificación es ampliamente aceptada como el problema central de diseño de un sistema de software complejo. Uno de los principios de las metodologías modernas de desarrollo de software es priorizar la definición, el diseño, la implementación y la evaluación de la arquitectura del software. La esencia de este principio es dedicar los mínimos esfuerzos a implementar un prototipo estable de arquitectura que garantice la viabilidad del proyecto en las fechas más tempranas posibles.

La arquitectura envuelve un conjunto de decisiones estratégicas de diseño, lineamientos, reglas y patrones que restringen el diseño y la implementación de un software. Aborda la problemática de más alto nivel de los proyectos, es crítica debido al alto riesgo de los proyectos actuales por su escala, distribución y complejidad. Determina la estructura general de un sistema, la cual debe satisfacer la calidad de servicio requerida, mitigar riesgos. Las decisiones de arquitectura restringen el diseño y la implementación de un proyecto.

La arquitectura de software es un concepto fácil de entender, a pesar de esto, es difícil de definir, resulta especialmente complicado diferenciar el diseño de la arquitectura. La arquitectura es un aspecto del diseño que se concentra en algunos factores especiales, no tiene que ver solamente con la estructura sino que es el nivel conceptual más alto del sistema y su ambiente. Además determina la adaptación del sistema a restricciones económicas, estilo e integridad del sistema. La estructura incluye grandes organizaciones, estructuras de control, protocolos de comunicación, sincronización, acceso a la información, asignación de funcionalidad a elementos de diseño, escalabilidad, performance y selección de diseño.

Virtudes de la arquitectura de software

Comunicación mutua.

La arquitectura de software representa un alto nivel de abstracción común que la mayoría de los participantes, si no todos, pueden usar como base para crear entendimiento mutuo, formar consenso y comunicarse entre sí. En sus mejores expresiones, la descripción arquitectónica expone las restricciones de alto nivel sobre el diseño del sistema, así como la justificación de decisiones arquitectónicas fundamentales.

Decisiones tempranas de diseño.

La arquitectura de software representa la encarnación de las decisiones de diseño más tempranas sobre un sistema, y esos vínculos tempranos tienen un peso fuera de toda proporción en su gravedad individual con respecto al desarrollo restante del sistema, su servicio en el despliegue y su vida de mantenimiento.

Restricciones constructivas.

Una descripción arquitectónica proporciona blueprints (planos, soluciones) parciales para el desarrollo, indicando los componentes y las dependencias entre ellos. Por ejemplo, una vista en capas de una arquitectura documenta típicamente los límites de abstracción entre las partes, identificando las principales interfaces y estableciendo las formas en que unas partes pueden interactuar con otras.

Reutilización, o abstracción transferible de un sistema.

La arquitectura de software encarna un modelo relativamente pequeño, intelectualmente tratable, de la forma en que un sistema se estructura y sus componentes se entienden entre sí; este modelo es transferible a través de sistemas; en particular, se puede aplicar a otros sistemas que exhiben requerimientos parecidos y puede promover reutilización en gran escala. El diseño arquitectónico soporta reutilización de grandes componentes o incluso de frameworks en el que se pueden integrar componentes.

Evolución.

La arquitectura de software puede exponer las dimensiones a lo largo de las cuales puede esperarse que evolucione un sistema. Haciendo explícitas estas “paredes” perdurables, quienes mantienen un sistema pueden comprender mejor las ramificaciones de los cambios y estimar con mayor precisión los costos de las modificaciones. Esas delimitaciones ayudan también a establecer mecanismos de conexión que permiten manejar requerimientos cambiantes de interoperabilidad, prototipado y reutilización.

Análisis.

Las descripciones arquitectónicas aportan nuevas oportunidades para el análisis, incluyendo verificaciones de consistencia del sistema, conformidad con las restricciones impuestas por un estilo, conformidad con atributos de calidad, análisis de dependencias y análisis específicos de dominio y negocios.

Administración.

La experiencia demuestra que los proyectos exitosos consideran una arquitectura viable como un logro clave del proceso de desarrollo industrial. La evaluación crítica de una arquitectura conduce típicamente a una comprensión más clara de los requerimientos, las estrategias de implementación y los riegos potenciales.

La arquitectura de software es importante porque:

- Facilita la comunicación entre todas las partes interesadas en el desarrollo de un sistema basado en computadora. Representa y destaca las decisiones más tempranas del diseño que tendrán un profundo impacto en todo el trabajo de la ingeniería de software que sigue, las más difíciles de modificar, las más críticas para ser tenidas en cuenta, las que sirven para lograr un lenguaje común y definir las guías principales del producto. Resulta vital en el éxito final del sistema como una entidad operacional, ya que allana el camino a la construcción de un sistema exitoso, lo contrario. Tiene un impacto en el rendimiento, en la facilidad de modificación del producto, la seguridad y confiabilidad.

CAPÍTULO 1

FUNDAMENTACIÓN TEÓRICA

La arquitectura de software es el marco fundamental para estructurar el sistema. Las propiedades de un sistema tales como rendimiento, seguridad y disponibilidad están influenciadas por la arquitectura utilizada. Las decisiones de diseño arquitectónico incluyen decisiones sobre el tipo de aplicación, distribución del sistema, el estilo arquitectónico a utilizar, patrones a aplicar, y las formas en las que la arquitectura debería documentarse y evaluarse.

Resulta importante tanto para el producto como para el proceso de desarrollo y construcción de software: priorizar la definición, el diseño, la implementación y la evaluación de la arquitectura de software, que es como se conoce al esqueleto o estructura del sistema. Desde el punto de vista de qué debe hacer el software, la arquitectura se define a partir de un conjunto de requisitos críticos funcionales, de rendimiento, o de calidad. Considerando cómo el software debe dar solución a tales objetivos, la arquitectura constituye el problema central de diseño, es decir, el conjunto de estructuras, clases y atributos principales del software y sus interfaces de comunicación.

Desde otro punto de vista más tangible, la arquitectura se materializa en el conjunto de componentes de código fuente y ejecutables que implementa dicho esqueleto, lo que posibilita demostrar y evaluar en qué medida el diseño da solución a aquellos requisitos críticos. Para proceder en este sentido puede aplicarse un proceso iterativo de prototipado, demostración y corrección, que permita atender y resolver en etapas iniciales del proyecto los riesgos asociados a los requisitos más críticos y a las decisiones de diseño más difíciles, que son aquellos que más pueden comprometer el éxito del proyecto. Así, el equipo de desarrollo debe diseñar, construir y estabilizar primero la arquitectura del software antes de diseñar e implementar el conjunto de componentes elementales que se integran en la arquitectura y que aportan las funcionalidades finales de usuarios.

La esencia del principio de priorizar la arquitectura es dedicar los mínimos esfuerzos a garantizar la corrección de las partes más importantes, costosas e indefinidas del sistema, y cuyo prototipo permita una demostración tangible de la viabilidad del proyecto.

Las características sobre las cuales hace énfasis la arquitectura son especialmente relevantes a la hora de pensar en: la evolución del sistema, performance, escalabilidad de la aplicación, disponibilidad, portabilidad y seguridad.

La estrategia de priorizar la arquitectura aporta significativos beneficios en materia de corrección al proceso de construcción del software, entre ellos:

- Evaluación de la solución (diseño + implementación) mediante demostraciones tangibles de sus capacidades desde fases muy tempranas de desarrollo
- Atención temprana a riesgos relacionados con la arquitectura que, generalmente, coinciden con aquellos que pueden conducir a mayores daños
- Propicia la construcción incremental del software (integración temprana) y las correspondientes actividades de verificación
- Propicia el desarrollo orientado a demostraciones periódicas de productos funcionales;
- Interfaces correctas permiten una cooperación eficiente entre diseñadores e implementadores.

Evaluación basada en demostración

La arquitectura es la materialización temprana, y de mínimo coste de las decisiones más importantes. El prototipo ejecutable de arquitectura debe permitir demostrar que la solución ya es madura, es decir, que tales decisiones son efectivas para resolver los principales casos de uso del software, en la cual un equipo de desarrolladores implementará e integrará la mayor parte del código. La arquitectura debe dar garantías de que la solución diseñada es realizable dentro de las restricciones de tiempo, personal, y presupuestos, o sea, que el proyecto es viable.

Atención a riesgos relacionados con la arquitectura

Un principio esencial de los métodos actuales de desarrollo de software es la gestión de riesgos relacionados con el proceso de desarrollo o con el producto de software. Este principio representa identificar al inicio del proyecto las dudas e incertidumbres que existen sobre qué hacer o cómo hacerlo, y definir planes para investigarlas y resolverlas. Si la arquitectura constituye los cimientos del software a construir, los riesgos relacionados con esta son lógicamente los más críticos del proyecto, es decir, aquellos que pueden producir los mayores errores o daños. Centrarse rápidamente en el diseño, implementación y estabilización de un prototipo de arquitectura implica tener que gestionar necesariamente dichos riesgos y resolverlos. Así se estaría eliminando las mayores causas potenciales de errores graves.

Integración temprana

Un tercer beneficio que se obtiene de la actividad de priorizar la arquitectura es que propicia una estrategia incremental de construcción del software como forma de aplicación del principio "divide y vencerás". Una arquitectura estable puede concebirse como una estructura de soporte del software, en la que se puedan ir integrando gradualmente las implementaciones de las diferentes funcionalidades finales. Cada nueva funcionalidad implementada e integrada es probada como unidad y como parte del todo en el que se inserta. Es decir, un nuevo requisito recién implementado puede ser inmediatamente validado desde la perspectiva del usuario en el contexto de la aplicación en el que se usará finalmente. La integración temprana permite dosificar los esfuerzos de realización de test de integración. Al integrar gradualmente cada funcionalidad en un prototipo de arquitectura previamente verificado y validado, la comprobación de la nueva funcionalidad dentro del prototipo se simplifica notablemente por reducirse las fuentes probables de error. Se deberá comprobar también que se mantiene la corrección de todo el prototipo en su interdependencia con la nueva funcionalidad.

Demostraciones periódicas a clientes y usuarios

El procedimiento incremental de construcción garantiza que en cada instante del proceso de construcción del software exista un prototipo funcional validado, aunque se encuentre en estado parcialmente terminado.

Cooperación eficiente a partir de interfaces de módulos

La interfaz de un módulo de software se refiere a la forma en la que dicho módulo interactúa con el resto del sistema: cómo se identifica, cómo se activa, qué datos necesita, y qué resultados devuelve. La especificación correcta de las interfaces de los módulos de la arquitectura propicia la cooperación y el paralelismo en la implementación de estas partes. El responsable de cada parte observará el resto de las partes como cajas negras disponibles con las que sabrá cómo interactuar. En general dichas partes podrán desarrollarse en paralelo e integrarse posteriormente.

1.2 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.

El arquitecto constituye un rol crítico en los proyectos, se focaliza en la calidad de servicio y lidera el proceso de definición y la implementación de la arquitectura. Reutiliza arquitecturas exitosas, framework y patrones de diseño. No es un diseñador en un proyecto. Adapta el sistema a las restricciones. Establece una estructuración correcta del sistema utilizando un conjunto de estrategias, herramientas y patrones de diseño. Es responsable de planificar, dirigir, seguir y controlar todas las tareas técnicas relacionadas con el desarrollo de software.

El principal reto del arquitecto de software es asegurar el éxito del proyecto diseñando las bases de la aplicación. Esto incluye la definición tanto de la estructura organizacional como de la estructura física de su despliegue. En este reto, el objetivo de los arquitectos es reducir la complejidad al dividir el sistema en partes simples, manejables y coherentes. La arquitectura resultante es extremadamente importante porque no solamente dicta como será construido el sistema más adelante, sino que permite establecer si la aplicación va a tener los atributos de calidad esenciales para un proyecto exitoso. Estos incluyen funcionalidad, fiabilidad, portabilidad, usabilidad, facilidad de mantenimiento, si cumple con los requerimientos de rendimiento y estándares de seguridad, y si puede evolucionar fácilmente ante los cambios de los requerimientos.

Beneficios de las habilidades del arquitecto para evaluación de tecnologías en proyectos:

Aceptación:
Conseguir la aceptación de los participantes del proyecto sustentando sus recomendaciones y decisiones con un buen nivel de fundamentación.

Uso adecuado: Asegurar el uso apropiado de las tecnologías por los miembros del equipo.

Como arquitecto se debe familiarizar con la mayor cantidad de patrones posibles:

- Dedique tiempo a los catálogos de patrones.
- Conozca los patrones que están disponibles.
- Adquiera experiencia en el uso de los patrones.
- Conozca como tipificar los problemas que resuelven los patrones.

Un arquitecto experimentado reutiliza:

1. Arquitecturas de sistemas exitosos.
2. Diseños que han funcionado correctamente en el pasado.
3. Frameworks conceptuales y tecnológicos.
4. Componentes y librerías.

Atributos del arquitecto de software

- Experiencia: en el dominio del problema, entendimiento de los requerimientos y de la problemática de la ingeniería de software.
- Liderazgo: con el fin de guiar los esfuerzos técnicos a través de varios equipos y para tomar decisiones críticas bajo presión.
- Comunicación: tener una buena capacidad de comunicación en aras ganarse la confianza del equipo, para persuadir, motivar y guiar. Debe interactuar tanto con el mundo exterior como con el resto del equipo. Juega el papel en la comunicación y coordinación en el área técnica.

El arquitecto debe tener conocimientos sólidos en la variedad de plataformas, frameworks, herramientas, patrones, lenguajes, en UML, en las técnicas de modelado de casos de uso, requerimientos del sistema, técnicas de modelado de sistemas tales como análisis y diseño orientado a objetos, así como tecnologías con las que el sistema va a ser implementado y desplegado.

RUP: el papel del arquitecto.

El arquitecto de software guía y coordina las actividades y los artefactos técnicos a través del desarrollo del proyecto. La metodología RUP define claramente las responsabilidades de este rol, el cual dirige el desarrollo de la arquitectura de software del sistema, que incluye la promoción y la creación de soporte para las decisiones técnicas clave que restringen el diseño global y la implementación para el proyecto.

Como resultado de la realización de varias de las actividades el arquitecto obtiene varios artefactos entre los que se destaca el documento de descripción de la arquitectura el cual proporciona una visión global de la arquitectura del sistema de software del proyecto, sirve como un medio de comunicación entre el arquitecto de software y demás miembros del equipo de proyecto respecto a las decisiones significativas para la arquitectura que se llevan a cabo, contiene los elementos más importantes para lograr la máxima abstracción en el diseño arquitectónico de la aplicación., ofrece un amplio panorama del sistema, utilizando una serie de vistas arquitectónicas para representar diferentes aspectos del sistema.

Además se exponen los estilos arquitectónicos para la aplicación, los principales componentes o elementos de la arquitectura, los conectores y sus configuraciones. Se describen los principales patrones de arquitectura utilizados, las restricciones de hardware o software de la arquitectura y se describe las tecnologías, herramientas que se utilizarán en el desarrollo del sistema, aspectos estructurales que incluyen la estructura global de control y la organización general; protocolos de comunicación, sincronización y acceso de datos; asignación de funciones para diseñar elementos; distribución física, composición de elementos de diseño; ajuste y rendimiento; y selección entre otras alternativas de diseño.

El documento tiene como objetivo estandarizar los elementos relacionados con la arquitectura de software principalmente con las vistas de la arquitectura, con el ambiente para la plataforma de desarrollo que se va a utilizar, patrones a emplear para la estructuración del producto de software, estándares de codificación; de modo que se convierta desde el primer momento en un documento normativo para el proceso de desarrollo de sistemas.

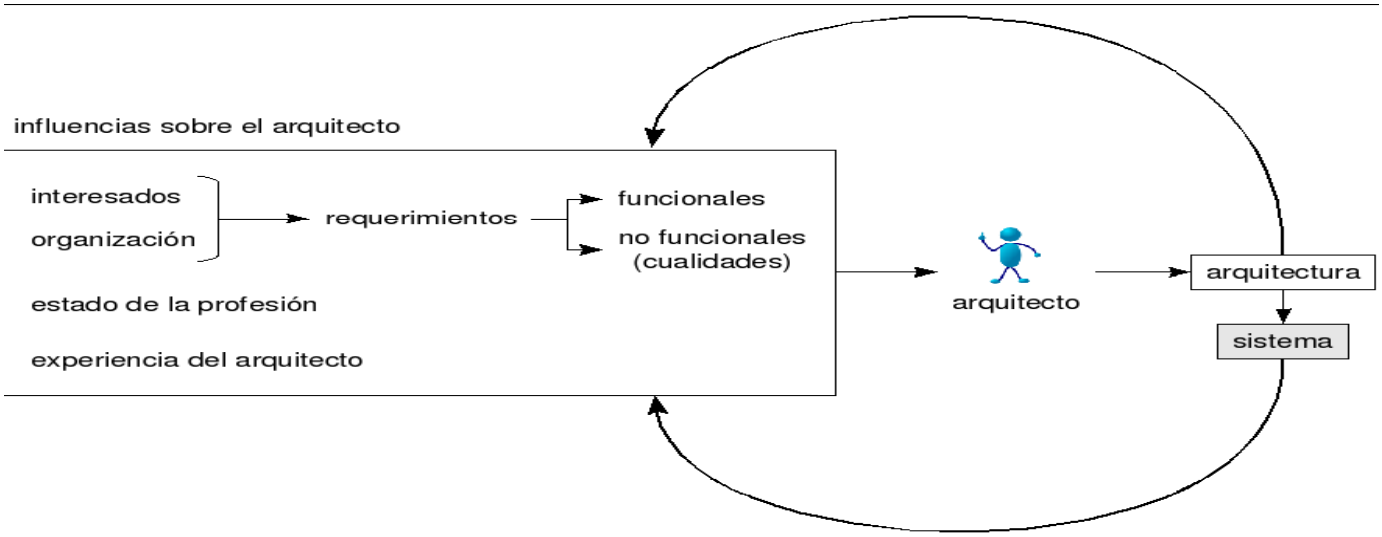


Fig. 1 Rol Arquitecto de Software

El diseño arquitectónico es un proceso en el que se intenta establecer una organización del sistema que satisfaga los requerimientos funcionales y no funcionales del propio sistema. Debido a que es un proceso creativo, las actividades dentro del proceso difieren radicalmente dependiendo del tipo de sistema a desarrollar, el conocimiento y la experiencia del arquitecto del sistema, y los requerimientos específicos del mismo. Basándose en su conocimiento y experiencia, los arquitectos del sistema tienen que responder a las siguientes cuestiones:

1. ¿Existe una arquitectura de aplicación genérica que pueda actuar como una plantilla para el sistema que se están diseñando?
2. ¿Cómo se distribuirá el sistema entre varios procesadores?
3. ¿Qué estilo arquitectónico es el apropiado para el sistema?
4. ¿Qué patrones son los adecuados para el sistema?
5. ¿Cuál será la aproximación fundamental utilizada para estructurar el sistema?
6. ¿Cómo se descompondrán en módulos las unidades estructurales del sistema?
7. ¿Qué estrategia se usará para controlar el funcionamiento de las unidades del sistema?
8. ¿Cómo se evaluará el diseño arquitectónico?
9. ¿Cómo debería documentarse la arquitectura del sistema?

1.3 Estilos arquitectónicos

Un estilo arquitectónico define las reglas generales de organización en términos de un patrón y las restricciones en la forma y la estructura de un grupo numeroso y variado de sistemas de software. En una forma más específica, un estilo determina el vocabulario de componentes y conectores que pueden ser utilizados en instancias de este estilo, con un conjunto de restricciones en las descripciones arquitectónicas.

El razonamiento sobre estilos de arquitectura es uno de los aspectos fundamentales de la disciplina. Un estilo es un concepto descriptivo que define una forma de articulación u organización arquitectónica. El conjunto de los estilos cataloga las formas básicas posibles de estructuras de software, mientras que las formas complejas se articulan mediante composición de los estilos fundamentales.

Los estilos arquitectónicos constituyen una generalización y abstracción de los patrones. Caracterizan una familia de sistemas que están relacionados por compartir propiedades estructurales y funcionales. Apuntan a describir sistemas completos y no partes de sistemas. Los estilos de arquitectura guían a la organización del sistema de software. Estos incluyen reglas y líneas a seguir para la organización de un sistema. Están vinculados a la forma más general en que están organizado un sistema de software, a las formas generales de la organización, mientras que los patrones están asociados a formas más concretas, que tienen que ver con la especialización. Los estilos expresan a la arquitectura de software en el nivel de abstracción más elevado, en el sentido más formal y teórico, mientras que los patrones se ocupan de cuestiones que están más cerca del diseño, la práctica, la implementación, el proceso, el refinamiento y el código.

Sirven para sintetizar estructuras de soluciones. Pocos estilos abstractos encapsulan una enorme variedad de configuraciones concretas. Definen los patrones posibles de las aplicaciones. Permiten evaluar arquitecturas alternativas con ventajas y desventajas conocidas ante diferentes conjuntos de requerimientos no funcionales.

Las arquitecturas complejas o compuestas resultan del agregado o la composición de estilos más básicos. Algunos estilos típicos son las arquitecturas basadas en flujo de datos, las peer-to-peer, las de invocación implícita, las jerárquicas, las centradas en datos o las de intérprete-máquina virtual.

Los estilos arquitectónicos describen un tipo de arquitectura de forma tal que la tipifica, la distingue y la singulariza. Dentro de cada uno de estos estilos se aplican patrones para regir la distribución de los elementos que lo componen tanto de hardware como de software, para seleccionar los subsistemas, paquetes, para diseñar las clases y demás actividades dentro del proceso de diseño arquitectónico. (6)

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. Ejemplares de la misma serían las arquitecturas de tubería-filtros y las de proceso secuencial en lote.

El estilo flujo de datos permite la transformación incremental de los datos. Arquitectura adecuada para aplicaciones que se entienden mejor en términos de datos que fluyen entre unidades de procesamiento y típica de los sistemas UNIX. Secuencia de procesos que realizan modificaciones incrementales de los datos. Cualquier proceso puede empezar su ejecución cuando empieza a recibir datos. Los sistemas de flujo de datos constituyen una sucesión de transformaciones de los datos de entrada.

Tubería y filtros

Una tubería es una arquitectura que conecta componentes computacionales (filtros) a través de conectores de modo que las computaciones se ejecutan a la manera de un flujo. Los datos se transportan a través de las tuberías entre los filtros, transformando gradualmente las entradas en salidas. El sistema tubería-filtros se percibe como una serie de transformaciones sobre sucesivas piezas de los datos de entrada. Los datos entran al sistema y fluyen a través de los componentes. La aplicación típica del estilo es un procesamiento clásico de datos: el cliente hace un requerimiento; el requerimiento se valida; un Webservice toma el objeto de la base de datos; se lo convierte a HTML; se efectúa la representación en pantalla. El estilo tubería-filtros propiamente dicho enfatiza la transformación incremental de los datos por sucesivos componentes.

Ventajas

Es simple de entender e implementar. Es posible implementar procesos complejos con editores gráficos de líneas de tuberías o con comandos de línea. Fuerza un procesamiento secuencial. Los filtros se pueden empaquetar, y hacer paralelos o distribuidos.

Desventajas

El patrón puede resultar demasiado simplista, especialmente para orquestación de servicios que podrían ramificar la ejecución de la lógica de negocios de formas complicadas. No maneja con demasiada eficiencia construcciones condicionales, bucles y otras lógicas de control de flujo. La independencia de los filtros implica que es muy posible la duplicación de funciones de preparación que son efectuadas por otros filtros (por ejemplo, el control de corrección de un objeto de fecha).

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. Permiten la manipulación compartida de datos. Los datos compartidos entre distintos clientes están en una estructura centralizada común. Esta familia de estilos enfatiza la integridad de los datos. Se estima apropiada para sistemas que se fundan en acceso y actualización de datos en estructuras de almacenamiento.

Arquitecturas de Pizarra o Repositorio

En esta arquitectura hay dos componentes principales: una estructura de datos que representa el estado actual y una colección de componentes independientes que operan sobre él. Un sistema de pizarra se implementa para resolver problemas en los cuales las entidades individuales se manifiestan incapaces de aproximarse a una solución, o para los que no existe una solución analítica, o para los que sí existen pero es inviable por la dimensión del espacio de búsqueda.

Ventajas

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

Desventajas

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

Estilo de Llamada y Retorno

El estilo llamada y retorno es el más utilizado en los grandes sistemas informáticos. Esta familia de estilos enfatiza la modificabilidad y la escalabilidad. Son los estilos más generalizados en sistemas en gran escala. Miembros de la familia son las arquitecturas de programa principal y subrutina, los sistemas basados en llamadas a procedimientos remotos, los sistemas orientados a objeto y los sistemas jerárquicos en capas. Los sistemas basados en llamada y retorno reflejan la estructura del lenguaje de programación.

Model-View-Controller (MVC)

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

Ventajas

Soporte de vistas múltiples. Dado que la vista se halla separada del modelo y no hay dependencia directa del modelo con respecto a la vista, la interfaz de usuario puede mostrar múltiples vistas de los mismos datos simultáneamente. Por ejemplo, múltiples páginas de una aplicación de Web pueden utilizar el mismo modelo de objetos, mostrado de maneras diferentes. Adaptación al cambio. Los requerimientos de interfaz de usuario tienden a cambiar con mayor rapidez que las reglas de negocios. Dado que el modelo no depende de las vistas, agregar nuevas opciones de presentación generalmente no afecta al modelo.

Desventajas

Complejidad. El patrón introduce nuevos niveles de indirección y por lo tanto aumenta ligeramente la complejidad de la solución. También se profundiza la orientación a eventos del código de la interfaz de usuario, que puede llegar a ser difícil de depurar.

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

Arquitecturas en Capas

Establece organización jerárquica tal que cada capa proporciona servicios a la capa inmediatamente superior y se sirve de las prestaciones que le brinda la inmediatamente inferior.

Ventajas

Soporta un diseño basado en niveles de abstracción crecientes, 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 que ellos llaman managers, debido a que son responsables de preservar la integridad de su propia representación. 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. Sólo soportan interacciones atómicas.

Arquitecturas Basadas en Componentes

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

Ventajas

Permite alcanzar un mayor nivel de reutilización de software. Permite que las pruebas sean ejecutadas probando cada uno de los componentes antes de probar el conjunto completo de componentes ensamblados. Simplifica el mantenimiento del sistema, cuando existe un débil acoplamiento entre componentes, el desarrollador es libre de actualizar y/o agregar componentes según sea necesario, sin afectar otras partes del sistema. Dado que un componente puede ser construido y luego mejorado continuamente por un experto u organización, la calidad de una aplicación basada en componentes mejorará con el paso del tiempo.

Desventajas

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

Estilos de Código Móvil

Esta familia de estilos enfatiza la portabilidad. Ejemplos de la misma son los intérpretes, los sistemas basados en reglas y los procesadores de lenguaje de comando.

Arquitectura de Máquinas Virtuales

Las máquinas virtuales simulan una funcionalidad no nativa del entorno. Se han llamado también intérpretes basados en tablas. Comprende básicamente dos formas o subestilos, que se han llamado intérpretes y sistemas basados en reglas.

Ventajas

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

Desventajas

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

Estilos heterogéneos

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

Sistemas de control de procesos

Desde el punto de vista arquitectónico, mientras casi todos los demás estilos se pueden definir en función de componentes y conectores, los sistemas de control de procesos se caracterizan no sólo por los tipos de componentes, sino por las relaciones que mantienen entre ellos. El objetivo de un sistema de esta clase es mantener ciertos valores dentro de ciertos rangos especificados, llamados puntos fijos o valores de calibración. La ventaja señalada para este estilo radica en su elasticidad ante perturbaciones externas.

Arquitecturas Basadas en Atributos

La intención de esta arquitectura es asociar a la definición del estilo arquitectónico un framework de razonamiento basado en modelos de atributos específicos. Su objetivo se funda en la premisa que dicha asociación proporciona las bases para crear una disciplina de diseño arquitectónico, tornando el diseño en un proceso predecible, antes que en una metodología. Los estilos basados en atributos incluyen atributos de calidad específicos que declaran el comportamiento de los componentes en interacción.

Estilos Peer-to-Peer

Esta familia, también llamada de 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 Recursos

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

Arquitecturas Basadas en Eventos

Las arquitecturas basadas en eventos se han llamado también de invocación implícita Los conectores de estos sistemas incluyen procedimientos de llamada tradicionales y vínculos entre anuncios de eventos e invocación de procedimientos. La idea dominante en la invocación implícita es que, en lugar de invocar un procedimiento en forma directa (como se haría en un estilo orientado a objetos) un componente puede anunciar mediante difusión uno o más eventos. El estilo se utiliza en ambientes de integración de herramientas, en sistemas de gestión de base de datos para asegurar las restricciones de consistencia, en interfaces de usuario para separar la presentación de los datos de los procedimientos que gestionan datos, y en editores sintácticamente orientados para proporcionar verificación semántica incremental.

Ventajas

Se optimiza el mantenimiento haciendo que procesos de negocios que no están relacionados sean independientes. Se alienta el desarrollo en paralelo, lo que puede resultar en mejoras de performance. Es fácil de empaquetar en una transacción atómica. Es agnóstica en lo que respecta a si las implementaciones corren sincrónica o asincrónicamente porque no se espera una respuesta. Se puede agregar un componente registrándolo para los eventos del sistema; se pueden reemplazar componentes.

Desventajas:

El estilo no permite construir respuestas complejas a funciones de negocios. Un componente no puede utilizar los datos o el estado de otro componente para efectuar su tarea. Cuando un componente anuncia un evento, no tiene idea sobre qué otros componentes están interesados en él, ni el orden en que serán invocados, ni el momento en que finalizan lo que tienen que hacer. Pueden surgir problemas de performance global y de manejo de recursos cuando se comparte un repositorio común para coordinar la interacción.

Arquitetura Orientada a Servicios (SOA)

Sólo recientemente estas arquitecturas que los conocedores llaman SOA han recibido tratamiento intensivo en el campo de exploración de los estilos. Es una relación entre servicios y consumidores de servicios, ambos lo suficientemente amplios como para representar una función de negocio completa.

Ventajas

Son varios los beneficios técnicos de una implementación de SOA entre ellos la conectividad, facilidad de mantenimiento, reducción de tamaño de proyectos, alta escalabilidad, reutilización real de los programas y mejora en tiempos de respuesta.

Esta arquitectura permite un alto grado de reutilización de los componentes desarrollados, debido a que “los servicios de aplicación son débilmente acoplados y altamente interoperables”, y las dependencias se establecen en tiempo de ejecución.

Desventajas

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

1.4 Patrones

El patrón es una descripción del problema y la esencia de su solución, de forma que la solución pueda reutilizarse en diferentes situaciones, no constituye una especificación detallada, es una solución adecuada a un problema común. Describe una solución probada a un problema recurrente de diseño, el cual ocurre en un contexto. Existen patrones de arquitectura, análisis y diseño. La definición de arquitectura requiere un proceso basado en el raciocinio sobre patrones. Un arquitecto reutiliza patrones para definir la estructura y el comportamiento interno y externo de un sistema.

Un patrón es una solución a un problema en un contexto, codifica conocimiento específico acumulado por la experiencia en un dominio. Un sistema bien estructurado está lleno de patrones. En el año 1977, Christopher Alexander expresa: "*Cada patrón describe un problema que ocurre una y otra vez en nuestro ambiente, y luego describe el núcleo de la solución a ese problema, de tal manera que puedes usar esa solución un millón de veces más, sin hacer jamás la misma cosa dos veces".*

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.

Proporcionan un esqueleto sobre el cual se implementan las funcionalidades deseadas en un sistema. Además de capturar la experticia del diseñador, permiten dar una respuesta a requisitos no funcionales tales como confiabilidad, extensibilidad, reutilización, facilidad de uso; en el sentido de proporcionar criterios o elementos para la selección.

Como un elemento en el mundo, cada patrón es una relación entre cierto contexto, cierto sistema de fuerzas que ocurre repetidas veces en ese contexto y cierta configuración espacial que permite que esas fuerzas se resuelvan. Como un elemento de lenguaje, un patrón es una instrucción que muestra la forma en que esta configuración espacial puede usarse, una y otra vez, para resolver ese sistema de fuerzas, donde quiera que el contexto la torne relevante.

El patrón es al mismo tiempo una cosa que pasa en el mundo y la regla dice cómo crear esa cosa y cuándo debemos crearla. Es tanto un proceso como una cosa; tanto una descripción de una cosa que está viva como una descripción del proceso que generará esa cosa. Entre los principales patrones se encuentran: (7)

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.

Patrones de Arquitectura

Un patrón arquitectónico define la estructura básica de una aplicación, provee un subconjunto de subsistemas predefinidos, incluyendo reglas, lineamientos para conectarlos y pautas para su organización y constituye una plantilla de construcción.

Los patrones arquitectónicos expresan una organización estructural para un sistema, permiten estructurar los componentes y subsistemas de un sistema. La abstracción más alta en cuanto a soluciones a través de patrones, se obtiene a través del uso de patrones de arquitectura. (8)

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.

Patrón arquitectónico MVC

El patrón 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.

Como su nombre lo dice, MVC consiste en separar lo mejor posible las capas de Modelo (el Sistema de Gestión de Base de Datos; los objetos que interactúan con la base de datos y efectúan los procesos pesados o “lógica de negocios”), la Vista (la presentación final de los datos procesados al cliente, comúnmente en formato HTML, y el código que provee de datos dinámicos a dichas páginas) y el Controlador (la capa que se encarga de recibir el input del usuario, delegar el trabajo a los Modelos apropiados e invocar las Vistas que correspondan; representa la separación clara entre el Modelo (lógica de negocio) y la Vista (interfaz gráfica), gracias a un controlador que los mantiene desacoplados)

Ventajas

- Se pueden mostrar distintas variantes de display simultáneamente, porque la vista no depende del modelo (ni el modelo de la vista)
- La interfaz tiende a cambiar más rápido que las reglas de negocios. Agregar nuevos tipos de vista no afecta al modelo
- El modelo es reusable con distintas vistas (ej.: una vista Web y una con interfaz de ventanas)
- División clara de trabajo entre los miembros de un equipo, que estará formado por personas con distintos niveles de especialización
- Soporte de vistas múltiples. Dado que la vista se halla separada del modelo y no hay dependencia directa del modelo con respecto a la vista, la interfaz de usuario puede mostrar múltiples vistas de los mismos datos simultáneamente.
- Adaptación al cambio. Los requerimientos de interfaz de usuario tienden a cambiar con mayor rapidez que las reglas de negocios. Los usuarios pueden preferir distintas opciones de representación, o requerir soporte para nuevos dispositivos. Dado que el modelo no depende de las vistas, agregar nuevas opciones de presentación generalmente no afecta al modelo.

Desventajas

- Puede aumentar un poco la complejidad de la solución. Como está guiado por eventos, puede ser algo más difícil de depurar.
- Si hay demasiados cambios en el modelo, la vista puede caer bajo un diluvio de refrescos, a menos que se prevea programáticamente (por ejemplo, actualizando varias modificaciones en lotes)

Consecuencias:

- Fácil de explicar y entender.
- Reduce dependencias y potencia la reutilización (cambiar interfaz sin tocar el resto).
- Permite dividir el trabajo en base a distintos roles (el diseñador gráfico no tiene por qué saber cómo se ha implementado el resto de componentes).
- No es “pura”: las dependencias cruzan varias capas.
- Complejidad (valorar costes).

Descripción del patrón

Para cada problema siempre hay buenas soluciones y para la programación Web, la solución más utilizada actualmente para organizar el código es el patrón de diseño MVC. En pocas palabras, el patrón MVC organiza el código en base a su función; separa el código en tres capas principales:

La **capa del modelo** define la lógica de negocio, la base de datos pertenece a esta capa. Encapsula los datos y las funcionalidades. Es independiente de cualquier representación de salida y/o comportamiento de entrada. Esta es la representación específica de la información con la cual el sistema opera. La lógica de datos asegura la integridad de estos y permite derivar nuevos datos.

La **vista** es lo que utilizan los usuarios para interactuar con la aplicación, los gestores de plantillas pertenecen a esta capa. En Symfony la capa de la vista está formada principalmente por plantillas en PHP. Este presenta el modelo en un formato adecuado para interactuar, usualmente la interfaz de usuario.

El **controlador** recibe las entradas, traducidas a solicitudes de servicio para el modelo. Es un bloque de código que realiza llamadas al modelo para obtener los datos y se los pasa a la vista para que los muestre al usuario. Este responde a eventos, usualmente acciones del usuario e invoca cambios en el modelo y probablemente en la vista.

Resumiendo:

1. El usuario interactúa con la interfaz de usuario de alguna forma (por ejemplo, el usuario pulsa un botón, enlace)
2. El controlador recibe la notificación de la acción solicitada y gestiona el evento.
3. El controlador accede al modelo, actualizándolo, posiblemente modificándolo de forma adecuada a la acción solicitada por el usuario
4. El controlador delega a los objetos de la vista la tarea de desplegar la interfaz de usuario. La vista obtiene sus datos del modelo para generar la interfaz apropiada para el usuario donde se refleja los cambios en el modelo (por ejemplo, produce un listado del contenido)

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.

Los patrones no se proponen descubrir ni expresar nuevos principios de la ingeniería de software, sino simplemente tratar de darle una representación a principios ya existentes que quizás ya han sido usado en los proyectos por un mero sentido de la lógica.

Un patrón de diseño es:

- ✓ Una solución estándar para un problema común de programación
- ✓ Una técnica para flexibilizar el código haciéndolo satisfacer ciertos criterios
- ✓ Un proyecto o estructura de implementación que logra una finalidad determinada
- ✓ Un lenguaje de programación de alto nivel
- ✓ Una manera más práctica de describir ciertos aspectos de la organización de un programa
- ✓ Conexiones entre componentes de programas
- ✓ La forma de un diagrama de objeto o de un modelo de objeto

Ventajas

- ✓ Proponen una forma de reutilizar la experiencia de los desarrolladores, para ello clasifica y describe formas de solucionar problemas que ocurren de forma frecuente en el desarrollo
- ✓ Por tanto, están basados en la recopilación del conocimiento de los expertos en desarrollo de software
- ✓ Es una experiencia real, probada y que funciona.
- ✓ Facilitan la localización de los objetos que formarán el sistema, la determinación de la granularidad adecuada, el aprendizaje y la comunicación entre programadores.
- ✓ Especifican interfaces para las clases e implementaciones al menos parciales.

Los patrones de diseño pretenden:

- Proporcionar catálogos de elementos reusables en el diseño de sistemas software.
- Evitar la reiteración en la búsqueda de soluciones a problemas ya conocidos y solucionados anteriormente.
- Formalizar un vocabulario común entre diseñadores.
- Estandarizar el modo en que se realiza el diseño.
- Facilitar el aprendizaje de las nuevas generaciones de diseñadores condensando conocimiento ya existente.

Así mismo, no pretenden:

- Imponer ciertas alternativas de diseño frente a otras.
- Eliminar la creatividad inherente al proceso de diseño.
- No es obligatorio utilizar los patrones siempre, solo en el caso de tener el mismo problema o similar que soluciona el patrón, siempre teniendo en cuenta que en un caso particular puede no ser aplicable. Abusar o forzar el uso de los patrones puede ser un error.

Elementos que caracterizan a los patrones:

Constituyen soluciones concretas debido a que proponen soluciones a problemas concretos, no son teorías genéricas. Son soluciones técnicas porque indican resoluciones técnicas basadas en Programación Orientada a Objetos (POO). En ocasiones tienen más utilidad con algunos lenguajes de programación y en otras son aplicables a cualquier lenguaje. Se utilizan en situaciones frecuentes ya que se basan en la experiencia acumulada la resolver problemas reiterativos. Favorecen la reutilización de código porque ayudan a construir software basado en la reutilización (a construir clases reutilizables). Los propios patrones se reutilizan cada vez que se vuelven a aplicar. El uso de un patrón no se refleja en el código ya que al aplicarlo, el código resultante no tiene por qué delatar el patrón o patrones que lo inspiró. No obstante, últimamente hay múltiples esfuerzos enfocados a la construcción de herramientas de desarrollo basados en los patrones y frecuentemente se incluye en los nombres de las clases el nombre del patrón en que se basan, facilitando así la comunicación entre desarrolladores. Es difícil reutilizar la implementación de un patrón porque al aplicarlo aparecen clases concretas que solucionan un problema concreto y que no será aplicable a otros problemas que requieran el mismo patrón.

Los patrones de diseño son abstracciones de alto nivel que documentan soluciones de diseño exitosas. Son fundamentales para reutilizar el diseño en el desarrollo orientado a objetos. Una descripción del patrón debería incluir: un nombre, que es una referencia significativa del patrón; una descripción del área del problema, que explica cuando puede aplicarse el patrón; una descripción de las partes de la solución del diseño, sus relaciones y sus responsabilidades; una declaración de los resultados y compromisos al utilizar el patrón, esto puede ayudar a comprender si un patrón puede ser aplicado de forma efectiva en una situación particular. Los patrones no debemos verlo como una moda, ni como normas, son mejores como intención, como núcleo de soluciones, como sugerencias.

La selección de los patrones depende de la estrategia que se pretenda seguir arquitectónicamente. Para adoptar el uso de un patrón, o la combinación de varios, es recomendable que antes se realice un estudio en detalle con respecto a la solución que plantean, sus beneficios, cuándo y cómo usarlo, qué variantes se pueden usar.

Patrones GRASP

GRASP es un acrónimo que significa General Responsibility Asignment Software Patterns (patrones generales de software para asignar responsabilidades) El nombre se eligió para indicar la importancia de captar estos principios, si se quiere diseñar eficazmente el software orientado a objetos.

Patrón Experto

Surge como principio fundamental que hay que tener en cuenta siempre cuando se esté asignando una responsabilidad a una clase. La respuesta es asignar la responsabilidad a la clase que contenga la información necesaria para cumplir la responsabilidad, o sea la clase debe ser la experta en la información. La responsabilidad de realizar una labor es de la clase que tiene o puede tener los datos involucrados (atributos).

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

El cumplimiento de una responsabilidad requiere a menudo de información distribuida en varias clases de objetos. Ello significa que hay muchos expertos "parciales" que colaboran en la tarea.

Los principales beneficios que se obtienen al usar este patrón son:

1. 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 al hecho de tener sistemas más robustos y de fácil mantenimiento.
2. 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 más fáciles de comprender y de mantener. Así se brinda soporte a una alta cohesión.

Patrón Creador

La creación de objetos es una de las actividades más frecuentes en un sistema orientado a objetos. En consecuencia, conviene contar con un principio general para asignar las responsabilidades concernientes a ella. Por tanto ¿Quién debería ser responsable de crear una nueva instancia de alguna clase?

Se asigna la responsabilidad de que una clase B cree un objeto de la clase A. La clase B crea las instancias de A si:

- B agrega los objetos de A
- B contiene los objetos de A
- B registra las instancias de los objetos de A.
- B tiene los datos de inicialización que serán enviados a la clase A cuando este objeto sea creado.

Este patrón es muy simple y su mayor beneficio es que contribuye a soportar el bajo acoplamiento, lo cual supone menos dependencias respecto al mantenimiento y mejores oportunidades de reutilización.

Patrón Bajo Acoplamiento

El acoplamiento es una medida de la fuerza con que una clase está relacionada a otras clases. Una clase con bajo o débil acoplamiento no depende de muchas otras. Por el contrario una clase con alto o fuerte acoplamiento recurre a muchas otras. Este tipo de clases no es conveniente, porque un cambio en las clases que utiliza ocasionaría cambios locales en la clase dependiente, además son más difíciles de entender cuando están aisladas y son más difíciles de reutilizar porque se requiere la presencia de otras clases de las que dependen.

Contexto o problema

¿Cómo dar soporte a una mínima dependencia y a un aumento de la reutilización?

- Una clase con bajo acoplamiento no depende de “muchas otras” clases.
- Las clases con alto acoplamiento recurren a muchas clases y no es conveniente.
- Son más difíciles de mantener, entender y reutilizar.

Debe haber pocas dependencias entre las clases. Es la idea de tener las clases lo menos ligadas entre sí que se pueda. De tal forma que en caso de producirse una modificación en alguna de ellas, se tenga la mínima repercusión posible en el resto de clases, potenciando la reutilización, y disminuyendo la dependencia entre las clases. La solución es asignar las responsabilidades de forma tal que las clases se comuniquen con el menor número de clases que sea posible, sin comprometer la funcionalidad por supuesto. El Bajo Acoplamiento soporta el diseño de clases más independientes, que reducen el impacto de los cambios, y también más reutilizables, que acrecientan la oportunidad de una mayor productividad, estas pueden considerarse sus mayores ventajas. Además este patrón no puede considerarse en forma independiente de otros patrones como Experto o el Creador, pues como ya vimos estos aportan principios comunes.

Alta Cohesión

Cada elemento del diseño debe realizar una labor única dentro del sistema, lo cual expresa que la información que almacena una clase debe de ser coherente y está en la mayor medida de lo posible relacionada con la clase.

En la perspectiva del diseño orientado a objetos, la cohesión es una medida de cuán relacionadas y enfocadas están las responsabilidades de una clase. Una clase con alta cohesión tiene responsabilidades estrechamente relacionadas y poco complejas. Una clase con baja cohesión hace muchas cosas no afines o un trabajo excesivo. Lo que traería algunos problemas como: que sean difíciles de comprender, difíciles de reutilizar, y difíciles de conservar; a estar clases son muy delicadas ya que las afectan constantemente los cambios. La solución es asignar a una clase responsabilidades que trabajen sobre una misma área de la aplicación y que no tengan mucha complejidad. Una regla práctica para medir este patrón es que una clase que consideremos con alta cohesión va a poseer un número relativamente pequeño, con una importante funcionalidad relacionada y poco trabajo por hacer. Colabora con otros objetos para compartir el esfuerzo si la tarea es grande. Este patrón brinda una mejor claridad y facilidad a la hora de entender el diseño, y simplifica el mantenimiento y las mejoras en funcionalidad.

Patrón Controlador

¿Quién debería encargarse de atender un evento del sistema?

Asignar la responsabilidad de controlar el flujo de eventos del sistema, a clases específicas.

Asignar la responsabilidad del manejo de mensajes de los eventos del sistema a una clase que represente alguna de las siguientes opciones:

- ✓ El sistema global.
- ✓ La empresa u organización global.
- ✓ Algo activo en el mundo real que pueda participar en la tarea.
- ✓ Un manejador artificial de todos los eventos del sistema de un caso de uso (controlador de casos de uso)

Además existen cuatro patrones GRASP adicionales:

Fabricación Pura.

Polimorfismo.

Indirección.

No hables con extraños.

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 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.

Respecto a su ámbito:

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

Patrones Creacionales

- **Abstract Factory (Fábrica abstracta):** Permite trabajar con objetos de distintas familias de manera que las familias no se mezclen entre sí y haciendo transparente el tipo de familia concreta que se esté usando. Proporciona una interfaz para crear familias de objetos o que dependen entre sí, sin especificar sus clases concretas.
- **Builder (Constructor virtual):** Abstrae el proceso de creación de un objeto complejo, centralizando dicho proceso en un único punto. Separa la construcción de un objeto complejo de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones.
- **Prototype (Prototipo):** Crea nuevos objetos clonándolos de una instancia ya existente. Especifica los tipos de objetos a crear por medio de una instancia prototípica, y crear nuevos objetos copiando este prototipo.

- **Factory Method (Método de fabricación):** Centraliza en una clase constructora la creación de objetos de un subtipo de un tipo determinado, ocultando al usuario la casuística para elegir el subtipo que crear. Define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan qué clase instanciar. Permite que una clase delegue en sus subclases la creación de objetos.
- **Singleton (Instancia única):** Garantiza la existencia de una única instancia para una clase y la creación de un mecanismo de acceso global a dicha instancia. Garantiza que una clase sólo tenga una instancia, y proporciona un punto de acceso global a ella.

Patrones Estructurales

- **Adapter (Adaptador):** Adapta una interfaz para que pueda ser utilizada por una clase que de otro modo no podría utilizarla. Convierte la interfaz de una clase en otra distinta que es la que esperan los clientes. Permiten que cooperen clases que de otra manera no podrían por tener interfaces incompatibles.
- **Bridge (Puente):** Desacopla una abstracción de su implementación. Desvincula una abstracción de su implementación, de manera que ambas puedan variar de forma independiente.
- **Composite (Objeto compuesto):** Permite tratar objetos compuestos como si de uno simple se tratase. Combina objetos en estructuras de árbol para representar jerarquías de parte-todo. Permite que los clientes traten de manera uniforme a los objetos individuales y a los compuestos.
- **Decorator (Envoltorio):** Añade dinámicamente nuevas responsabilidades a un objeto, proporcionando una alternativa flexible a la herencia para extender la funcionalidad.
- **Facade (Fachada):** Provee de una interfaz unificada simple para acceder a una interfaz o grupo de interfaces de un subsistema. Proporciona una interfaz unificada para un conjunto de interfaces de un subsistema. Define una interfaz de alto nivel que hace que el subsistema se más fácil de usar.
- **Flyweight (Peso ligero):** Reduce la redundancia cuando gran cantidad de objetos poseen idéntica información. Usa el compartimiento para permitir un gran número de objetos de grano fino de forma eficiente.
- **Proxy:** Mantiene un representante de un objeto. Proporciona un sustituto o representante de otro objeto para controlar el acceso a éste.

Patrones de Comportamiento

- **Chain of Responsibility (Cadena de responsabilidad):** Permite establecer la línea que deben llevar los mensajes para que los objetos realicen la tarea indicada. Evita acoplar el emisor de una petición a su receptor, al dar a más de un objeto la posibilidad de responder a la petición. Crea una cadena con los objetos receptores y pasa la petición a través de la cadena hasta que esta sea tratada por algún objeto.
- **Command (Orden):** Encapsula una operación en un objeto, permitiendo ejecutar dicha operación sin necesidad de conocer el contenido de la misma. Encapsula una petición en un objeto, permitiendo así parametrizar a los clientes con distintas peticiones, encolar o llevar un registro de las peticiones y poder deshacer la operaciones.
- **Interpreter (Intérprete):** Dado un lenguaje, define una gramática para dicho lenguaje, así como las herramientas necesarias para interpretarlo y define una representación de su gramática junto con un intérprete que usa dicha representación para interpretar las sentencias del lenguaje.
- **Iterator (Iterador):** Permite realizar recorridos sobre objetos compuestos independientemente de la implementación de estos. Proporciona un modo de acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación interna.
- **Mediator (Mediador):** Define un objeto que coordine la comunicación entre objetos de distintas clases, pero que funcionan como un conjunto, objeto que encapsula cómo interactúan un conjunto de objetos. Promueve un bajo acoplamiento al evitar que los objetos se refieran unos a otros explícitamente, y permite variar la interacción entre ellos de forma independiente.
- **Memento (Recuerdo):** Permite volver a estados anteriores del sistema. Representa y externaliza el estado interno de un objeto sin violar la encapsulación, de forma que éste puede volver a dicho estado más tarde.
- **Observer (Observador):** Define una dependencia de uno-a-muchos entre objetos, de forma que cuando un objeto cambie de estado se notifique y actualicen automáticamente todos los objetos que dependen de él.
- **State (Estado):** Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno. Parecerá que cambia la clase del objeto.

- **Strategy (Estrategia):** Permite disponer de varios métodos para resolver un problema y elegir cuál utilizar en tiempo de ejecución. Define una familia de algoritmos, encapsula uno de ellos y los hace intercambiables. Permite que un algoritmo varíe independientemente de los clientes que lo usan.
- **Template Method (Método plantilla):** Define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos, esto permite que las subclases redefinan ciertos pasos de un algoritmo sin cambiar su estructura.
- **Visitor (Visitante):** Permite definir nuevas operaciones sobre una jerarquía de clases sin modificar las clases sobre las que opera. Representa una operación sobre los elementos de una estructura de objetos. Permite definir una nueva operación sin cambiar las clases de los elementos sobre los que opera.

¿Qué beneficios produce la utilización de patrones de diseño?

- Contribuyen a reutilizar diseño, identificando aspectos claves de la estructura de un diseño que puede ser aplicado en una gran cantidad de situaciones. La importancia de la reutilización del diseño no es despreciable, ya que ésta provee de numerosas ventajas: reduce los esfuerzos de desarrollo y mantenimiento, mejora la seguridad, eficiencia y consistencia del diseño, y proporciona un considerable ahorro en la inversión.
- Mejoran (aumentan, elevan) la flexibilidad, modularidad y extensibilidad, factores internos e íntimamente relacionados con la calidad percibida por el usuario.
- Incrementan el vocabulario del diseño, permitiendo diseñar desde un mayor nivel de abstracción.

1.5 Lenguaje de descripción de la arquitectura

Los ADLs (Architectecture Description Language) surgen como una respuesta a la necesidad de representar los sistemas de software desde el punto de vista arquitectónico a un alto nivel de abstracción, de manera que tales descripciones puedan ser utilizadas en las primeras etapas de desarrollo como una herramienta para conocer y evaluar el comportamiento global de la arquitectura propuesta para un sistema en función de los requisitos. Existen ADLs de propósito general y otros de dominio específicos.

Un ADL es un lenguaje o notación para describir una arquitectura de software; realizar una descripción de componentes, conectores y enlaces entre ellos; una herramienta para la verificación de la arquitectura y el prototipo. Constituyen un instrumento para lograr una mayor formalización en las descripciones arquitectónicas, permitiendo que dichas descripciones adquieran una semántica bien definida manteniendo al mismo tiempo un equilibrio un elevado nivel de abstracción.

Los ADLs permiten modelar una arquitectura mucho antes que se lleve a cabo la programación de las aplicaciones que la componen, analizar su adecuación, determinar sus puntos críticos y eventualmente simular su comportamiento. Diseñados específicamente para describir la arquitectura de los sistemas de software: componentes, que no son más que elementos computacionales y de datos descritos mediante los roles que juegan; los conectores, mecanismos de interacción flexibles y variados; y las configuraciones, combinación de componentes y conectores interconectados.

Ejemplos: Wright, Darwin, UniCon, Rapide, C2, LEDA.

- UNICON (Shaw et al. 1995)
- Rapide (Luckham et al. 1995)
- Darwin (Magee y Kramer, 1996; 1999)
- Wright (Allen y Garlan, 1997; 1998)
- Executable Connectors (Ducasse y Richner, 1997)

Problema: No cubren todo el ciclo de vida de las aplicaciones software (sólo diseño preliminar), no llegan a la implementación.

El modelado de la arquitectura de software usando ADLs

Los ADLs facilitan la construcción de modelos de alto nivel en los cuales los sistemas son descritos como composiciones de componentes. Juegan un papel importante en el desarrollo de software, influyendo en la calidad ya que permite razonar acerca de las propiedades estructurales en el desarrollo de procesos; facilitando la producción de estructuras extensibles, localizar desperfectos del diseño y mantener la consistencia.

Requisitos de un ADL

Composicional: Basado en la descripción de interfaces.
Dinamismo: Sistemas con configuración o topología cambiante.
Verificación de propiedades: Fundamento formal que permita el análisis.
Proceso de desarrollo.

→ Mecanismos de refinamiento y parametrización.
→ Reutilización de componentes y arquitecturas.
→ Simulación y generación del sistema final.

Características de los ADLs

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

Una de las características deseables en un ADL, es la capacidad de ofrecer distintas vistas de una misma arquitectura compleja. Los elementos básicos de los ADLs son componentes y conectores, e incluyen reglas y recomendaciones para arquitecturas bien formadas. Sin embargo, como todos los lenguajes especializados, los ADLs solamente pueden ser comprendidos por expertos del lenguaje y son inaccesibles para los especialistas tanto de las aplicaciones como del dominio. Esto hace que sea difícil analizarlos desde una perspectiva práctica. Las notaciones como UML constituyen las más comúnmente usadas para la descripción arquitectónica.

La forma que RUP propone para la descripción de la arquitectura es mediante la vista de los modelos de casos de uso, análisis, diseño, implementación y despliegue del sistema, o sea se describe las partes del sistema que es importante que comprendan todos los desarrolladores e interesados del sistema en general. Estas vistas se desarrollan utilizando el Lenguaje Unificado de Modelado (UML) aunque este no es un ADL.

UML es un lenguaje de modelado visual que se usa para especificar, visualizar, construir y documentar artefactos de un sistema de software. Captura decisiones y conocimiento sobre los sistemas que se deben construir. Se usa para entender, diseñar, hojear, configurar, mantener, y controlar la información sobre tales sistemas. Está pensado para usarse con todos los métodos de desarrollo, etapas del ciclo de vida, dominios de aplicación y medios.

UML ha mejorado el desarrollo de software no sólo al establecer un estándar común que simplifica la comunicación entre desarrolladores de software. Es utilizado no sólo para la especificación de un sistema sino también para propósitos de comunicación entre la gente involucrada en el desarrollo de un sistema o para la documentación de software existente. Como resultado de las ventajas que ofrece UML, se seleccionó como lenguaje de modelado. Además, por poseer un propósito general y comprensible, se propone para describir la arquitectura, a pesar de no ser propiamente un ADL.

El lenguaje de modelado pretende unificar la experiencia pasada sobre técnicas de modelado e incorporar las mejores prácticas actuales en un acercamiento estándar. UML incluye conceptos semánticos, notación, y principios generales. Tiene partes estáticas, dinámicas, de entorno y organizativas. La especificación de UML no define un proceso estándar pero está pensado para ser útil en un proceso de desarrollo iterativo. Pretende dar apoyo a la mayoría de los procesos de desarrollo orientados a objetos. (9)

UML es un lenguaje: proporciona un vocabulario y unas reglas que se centran en la representación conceptual y física de un sistema, y que indican cómo crear y leer modelos bien formados. Sin embargo, no dice qué modelos crear ni cuándo se deberían crear, ésta es la tarea del proceso de desarrollo de software.

UML es un lenguaje para visualizar: es un lenguaje gráfico que mezcla gráficos y texto, pero es algo más que un simple montón de símbolos. De hecho, detrás de cada símbolo en la notación UML hay una semántica bien definida, de manera que un desarrollador puede escribir un modelo en UML, y otro desarrollador, o incluso otra herramienta, puede interpretar ese modelo sin ambigüedad.

UML es un lenguaje para especificar: cubre la especificación de todas las decisiones de análisis, diseño e implementación que deben realizarse al desarrollar y desplegar un sistema con gran cantidad de software.

UML es un lenguaje para construir: no es un lenguaje visual, pero sus modelos pueden conectarse de forma directa con una gran variedad de lenguajes de programación. Es posible establecer correspondencias desde un modelo UML a un lenguaje de programación como Java o C++, o incluso a tablas en una base de datos relacional o al almacenamiento persistente en una base de datos orientada a objetos. Permite ingeniería directa e inversa.

UML es un lenguaje para documentar: cubre toda la documentación de la arquitectura de un sistema y todos sus detalles. También proporciona un lenguaje para expresar requisitos y pruebas del software. Finalmente, UML proporciona un lenguaje para modelar las actividades de planificación de proyectos y gestión de versiones.

1.6 Conclusiones parciales del capítulo

En el presente capítulo se han abordado los aspectos relacionados con el objeto de estudio, con el objetivo de proveer una solución que permita la gestión de la información y conformar un Sistema Integral de Gestión para la Vigilancia de Equipos Médicos. Una vez plasmado y profundizado en el presente capítulo el estudio sobre los aspectos referentes al tema central de la investigación, los que ayudarán al desarrollo del trabajo tales como: la arquitectura de software, los principales estilos arquitectónicos, patrones tanto de arquitectura como de diseño, el rol arquitecto de software dentro de la producción de software y los lenguajes de descripción de arquitectura; tales aspectos dan paso a las acciones del capítulo 2 y sirven de base para proporcionar una solución simple, escalable y confiable, definir en el grupo una arquitectura sólida, robusta y bien concebida, así como coordinar la estandarización e integración de los módulos y aplicaciones. Finalmente se propone por las características del sistema y las necesidades del grupo de desarrollo seguir la línea del estilo arquitectónico llamada y retorno, el patrón de arquitectura MVC, se propone aplicar los patrones de diseño GOF y GRASP y UML como lenguaje para describir la arquitectura, independientemente de que no constituye un ADL.

CAPÍTULO 2: ARQUITECTURA DEL SISTEMA

Introducción

El capítulo en cuestión se dedica a la plataforma de desarrollo, plataforma de despliegue, requerimientos de hardware, estructura y organización del proyecto. Así como a representar la descripción de la arquitectura de software a través de las vistas de la arquitectura.

El problema de la descripción de una arquitectura de software es encontrar una técnica que cumpla con los propósitos del desarrollo de software, en otras palabras la comunicación entre las partes interesadas, la evaluación y la implementación. Sería bueno antes de todo, responder la siguiente interrogante: ¿Cómo diseñar una arquitectura robusta y flexible a la vez, que permita responder a las necesidades del control de indicadores de refinerías?

Diseñar una arquitectura de software para el Grupo de Desarrollo para la Gestión de Equipos Médicos que permita al equipo tomar decisiones sobre los aspectos dinámicos y estáticos más significativos de la organización del sistema, la selección de elementos estructurales mediante los cuales se compone dicho sistema, sus interfaces, comportamiento; y que permita al equipo conocer la estructura del sistema que se está desarrollando.

Para definir la arquitectura del sistema es necesario contar con todos los requerimientos, aunque en la práctica esto no ocurre así, ya que se ha convertido en una utopía el simple hecho de poder contar con el 100% de los requerimientos, debido a que estos van apareciendo durante toda la vida del sistema o todo el ciclo del proceso de desarrollo de software. Los requerimientos constituyen sin lugar a dudas una influencia determinante en la concepción de la arquitectura del sistema. Constituye una necesidad que el sistema verifique algunos requerimientos no funcionales o cualidades del sistema tales como modificabilidad, seguridad, desempeño, tolerancia a fallas, testeabiblidad, entre otros y desde luego comprobar varias cualidades simultáneamente. Cualidades estas que se deben incorporar conscientemente en la arquitectura del sistema desde la concepción debido a que un requerimiento no funcional suele ser muchísimo más costoso que no tener en cuenta un requisito funcional, ambos requisitos deben guiar la definición de la arquitectura del sistema.

Tomando este punto de partida el trabajo continúa con la elección del estilo arquitectónico y el patrón o la combinación de patrones de arquitectura, luego definir el conjunto de patrones de diseño que se aplicarán y finalmente con todas estas armas se podrá pasar a diseñar los módulos del sistema.

El nivel arquitectónico de la estructura de un sistema es aquella descripción donde se utilizan conectores diferentes a llamada a procedimiento o se imponen restricciones importantes entre los componentes o aparecen distintos tipos de componentes en la descripción.

La descripción del nivel arquitectónico implica el uso de elementos de software que no tienen representación directa en la mayoría de los lenguajes de programación, es decir, los elementos abstractos con los cuales trabaja el arquitecto y que los programadores deberán refinar y proyectar sobre la tecnología de implementación disponible.

En el nivel arquitectónico los conectores son tratados como elementos de primer nivel esto implica entender que una buena arquitectura depende tanto de las relaciones entre los componentes como de los componentes en sí. El diseño de la arquitectura de software se ha convertido en un paso indispensable en el desarrollo de grandes proyectos de software.

La descripción de la arquitectura es una representación de alto nivel de la estructura de un sistema o aplicación que describe los componentes que lo integran, interacciones entre ellos, patrones que supervisan su composición así como las restricciones para aplicar dichos patrones.

En este proceso de definir la arquitectura de software imperan una serie de motivos tales como:

- → Se pueden evaluar las decisiones tempranas a la hora de construir el software.
- → Alto nivel de abstracción.
- → Vinculada con requerimientos no funcionales.
- → Fuerte impulso en la academia y la industria.
- → Fomenta la reutilización y aumenta calidad en la producción.

Alcanzar una arquitectura estable que aporte garantías sobre la viabilidad del proyecto, se considera el punto de transición entre lo que se suele denominar la fase de ingeniería (definición del producto y su solución) y la fase de producción (construcción, integración, evaluación y entrega del producto). En la primera se toman las decisiones más importantes mientras que en la segunda se realizan los mayores gastos y esfuerzos.

La clave del éxito y a su vez la dificultad del principio de priorizar la arquitectura consiste en definir qué es y qué no es la arquitectura. Si se incluyen demasiados detalles se puede correr el riesgo de la propiedad de configuración mínima necesaria (o más simple posible), que permite demostrar, con el mínimo esfuerzo y en fechas tempranas, la corrección de la solución diseñada. Si, por el contrario, se define una configuración más simple de lo necesario, probablemente se estaría ignorando los requisitos, riesgos e interacciones críticas, lo cual impide dar garantías sobre el éxito del proyecto antes de realizar los mayores esfuerzos y gastos de la fase de producción.

Desde un punto de vista de gestión del proceso de desarrollo, se pueden identificar dos dimensiones para manejar la arquitectura:

• **Descripción de arquitectura:** subconjunto del modelo de diseño del software. Incluye elementos significativos de la arquitectura y excluye el diseño de las componentes básicas. Resuelve decisiones sobre qué desarrollar, qué reutilizar y qué comprar.

• **Versión estable de arquitectura:** subconjunto suficiente de componentes ejecutables que permiten demostrar lo antes posible que el método de solución (diseño, tecnología, tiempo y costes estimados) puede resolver satisfactoriamente la definición del producto (objetivos, alcance, rendimiento, beneficios, calidad). Desde una perspectiva técnica, como se ha comentado antes, la arquitectura engloba requisitos críticos, decisiones de diseño, componentes de código fuente, y componentes ejecutables, información que debe ser modelada e incluida en el documento de descripción de arquitectura.

2.1 Organización del sistema

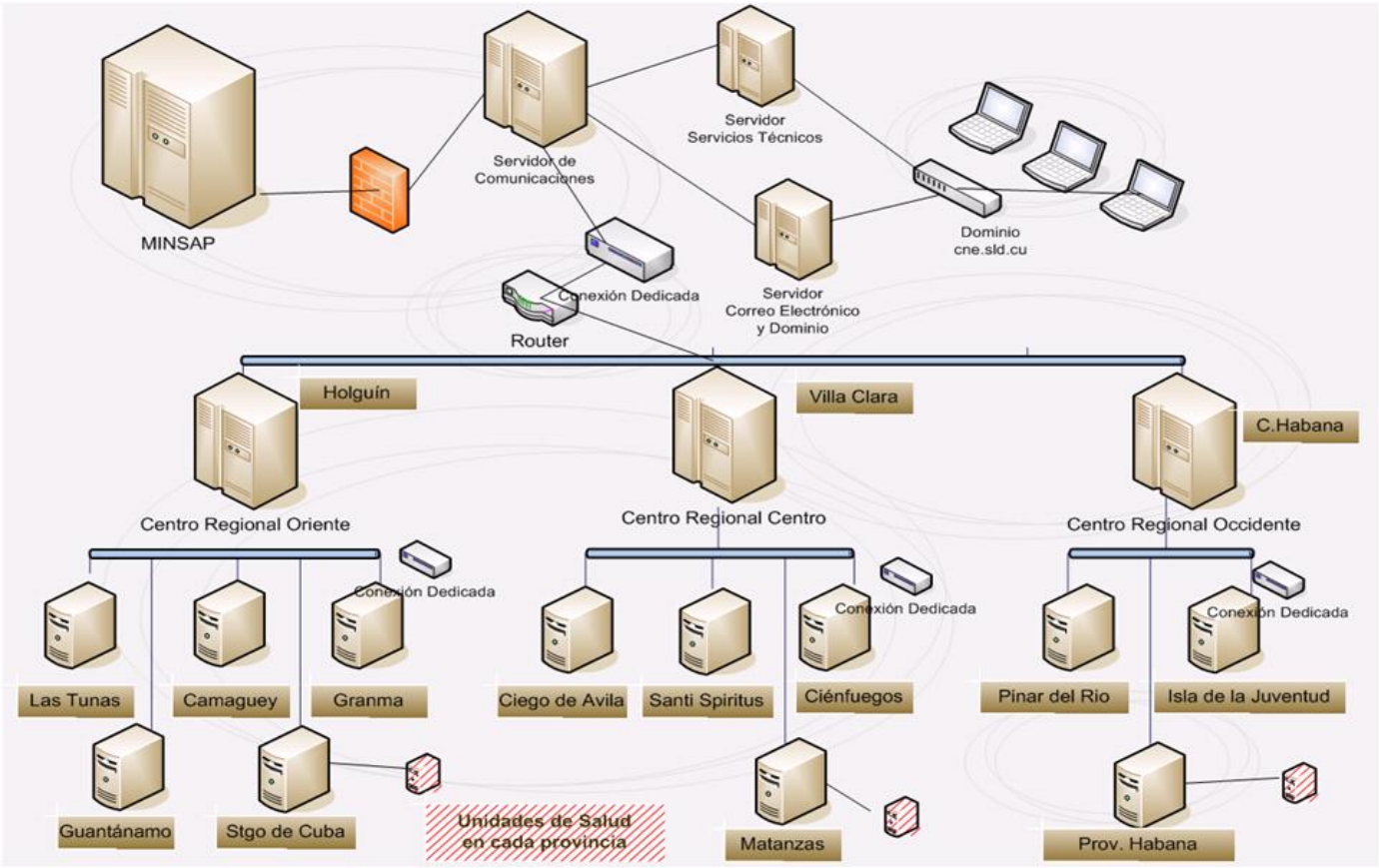


Fig. 2 Organización General del Sistema

2.2 Objetivos, metas y restricciones arquitectónicas

El objetivo de la arquitectura es lograr un sistema escalable, configurable y portable. A continuación se mencionan los requerimientos y restricciones del sistema que tienen un impacto significativo en la arquitectura:

- ✓ Se deben recoger los datos de todas las piezas de los equipos médicos por modelo que existen en el país.
- ✓ Se deben recopilar los datos de de todos los especialistas de electromedicina del país.
- ✓ Los datos que suministran los especialistas pueden estar en copia dura (documentos impresos) o en formato digital.
- ✓ Los datos con los cuales operan ambos sistemas son de vital importancia y de carácter estratégico para la nación por lo cual es necesario garantizar su seguridad.
- ✓ Existirá un servidor central y tres regionales en los cuales se van a almacenar todos los datos recopilados y en cada unidad de trabajo tendrán copias de sus datos.
- ✓ Interoperabilidad con sistemas legados.
- ✓ Uso de estándares de codificación y de base de datos en el grupo.
- ✓ Idioma: Se debe utilizar como idioma el español, las palabras no se acentuarán.

Los proyectos que se coordinan con entidades del SNS dentro y fuera del país están en la obligación de reconocer y cumplir con las políticas e intereses del MINSAP y admitir la evaluación y certificación de las soluciones informáticas para el sector de la salud pública. Entre las principales políticas resaltan: (10)

- Deben constituirse en componentes modulares y estables, que compartan normas y cooperen entre sí.
- La informatización debe alinearse con las tecnologías de punta y los estándares de calidad desarrollados en el mundo y adecuarse a las condiciones particulares.
- La seguridad informática y de contingencia son requisitos imprescindibles y responsabilidad ineludible de los productores, prestadores y usuarios.

Los requisitos no funcionales que se describen a continuación constituyen la base que debe sustentar la arquitectura, deben responder a las políticas de informatización en el sector de la salud y deben ser de estricto cumplimiento por parte de las entidades involucradas en el desarrollo de un producto informático para el SNS. La reutilización de dicha arquitectura garantizará el aprovechamiento de los recursos tecnológicos con los que se cuenta. (10)

RNF 1. Requerimientos de seguridad

El sistema deberá contar con un mecanismo de seguridad basado en el modelo de Autenticación, Autorización y Auditoria (Authentication, Authorization and Accounting, AAA) con Autenticación de firma única (Single Sign On).

RNF 1.1 La autenticación será la primera acción del usuario en el sistema y consistirá en suministrar un nombre de usuario único y una contraseña que debe ser de conocimiento exclusivo de la persona que se autentica. Si el usuario autenticado no se encuentra registrado se debe reportar un error de acceso.

RNF 1.2 Las trazas del sistema contendrán un texto descriptivo de las acciones realizadas así como el día, mes, año, hora, minuto, segundo en que se registra. Cada Petición de usuario a un componente, autorizada o no, será registrada en las trazas del sistema. Cada componente, podrá registrar la información que considere deba formar parte de la traza del sistema.

RNF 1.3 El sistema permitirá a las personas designadas como administradores establecer las políticas de seguridad en su nivel de acceso. Debe existir al menos un administrador por cada nivel. Los administradores podrán crear, cuentas de administradores en los niveles inferiores y cuentas de usuarios Editores o Visualizadores en su nivel.

RNF 2. Requerimientos de confiabilidad

El sistema deberá ser capaz de prevenir o recuperarse ante posibles fallos.

RNF 2.1 La información manejada por el sistema será objeto de cuidadosa protección contra la corrupción y estados inconsistentes. Deberán existir mecanismos de chequeo de integridad.

RNF 2.2 Deberá existir una estrategia de replicación que permita, de manera transparente para el usuario final, balancear la carga de acceso entre múltiples servidores aumentando los tiempos de respuesta y facilitar la recuperación inmediata del sistema si falla uno de ellos.

RNF 2.3 Se permitirá la creación de copias de respaldo que puedan restaurar el sistema en caso de fallo crítico o pérdida total de la información.

RNF 3. Requisito de interfaz interna.

RNF 3.1 La información que puede ser reutilizable entre varios sistemas deberá ser desarrollada como XML Web Services.

RNF 4. Requerimiento de software.

RNF 4.1 Todo el software debe correr sobre el Sistema Operativo Linux.

RNF 4.2 El software base debe responder a las políticas de Software Libre y de Fuente Abierta.

RNF4.3 Los clientes tendrán acceso al sistema a través de cualquier navegador Web, o sea para la navegación del sistema se realizará con el uso de navegadores: Internet-Explorer, Mozilla Firefox, Opera, Netscape u otro compatible, preferentemente el Mozilla Firefox.

RNF 5. Requerimiento de hardware.

Requerimientos mínimos para el servidor:

- Microprocesador Pentium superior a 2.0 GHz
- 512 MB RAM o superior.
- 4 GB de espacio en disco como mínimo.

Requerimientos mínimos para la conexión del cliente:

- Microprocesador Pentium superior a 333 MHz.
- 256 MB RAM o superior.
- MODEM o red con TCP-IP para conexión al servidor

RNF 6. Requisito de diseño y la implementación.

RNF 6.1 Todos los componentes del sistema deben desarrollarse siguiendo el principio de máxima cohesión y mínimo acoplamiento.

Cohesión: es una medida de la fuerza relativa funcional de un componente. Un componente con cohesión realiza una sola tarea dentro de un procedimiento de software, requiriendo poca interacción con los otros componentes. Un componente con cohesión alta debería hacer una sola función.

Acoplamiento: es una medida de la interdependencia relativa entre los componentes. Minimizando el acoplamiento se evita el efecto "onda" en la propagación de errores.

Elementos relativos a la organización y presentación (11)

- La organización de la información debe basarse en las características de los usuarios, sus necesidades, sus capacidades tecnológicas, demandas, etc.
- Seguir la regla de los tres clic, la cual plantea que si se tiene que hacer más de tres clic sobre un anclaje (hipervínculo) para acceder a la información pertinente provoca cansancio y desprecio, o sea si un sistema no enseña en pocos minutos no es interesante.
- Se debe organizar la información para que se escalable, es decir que el producto pueda crecer sin tener grandes cambios en la estructura y funcionamiento del mismo.
- El producto debe informar sus objetivos a primera vista.
- No se debe abusar de las imágenes animadas, debido a que el uso excesivo de ellas o la no coherencia de las mismas provoca un caos visual dentro del producto.
- Es importante distinguir el estado del objeto o vínculo sobre el que se ejecutó la acción.

Estándares generales de base de datos y estándares de codificación

Para el desarrollo del sistema quedaron definidos los estándares generales de base de datos y los estándares de codificación separados ambos en dos documentos oficiales dentro del grupo. Este aspecto es significativo dentro de la arquitectura por lo que se hace alusión y referencia al contenido de dichos documentos.

El uso de ellos partiendo de las convenciones definidas permite una mejor comunicación entre los diseñadores de base de datos, programadores y desarrolladores creando las condiciones para la reusabilidad y el mantenimiento de los sistemas. Para definir el estilo de codificación a seguir en la aplicación se utilizó la notación estándar establecida para aplicaciones desarrolladas en PHP (PHP Coding Standard), que mayormente está basada en el estándar de código para aplicaciones en C++ (C++ Coding Standard). Los estándares generales de base quedaron bien explícitos cada uno de los aspectos fundamentales con una breve descripción y ejemplos para una rápida adaptación y comprensión por parte del equipo de desarrollado. [Ver artefactos definidos en el grupo: Estándares de Codificación y Estándares de Base de Datos].

2.3 Plataforma tecnológica

Plantear un soporte al desarrollo del sistema con el uso de herramientas informáticas que faciliten la construcción del mismo, y definir la plataforma tecnológica a utilizar, constituye una de las buenas prácticas de la arquitectura de software. Para precisar dicha plataforma se deben tomar un conjunto de decisiones técnicas que restringen el diseño y la implementación del sistema.

La prospección tecnológica sobre herramientas, librerías, frameworks, estilos, patrones y prácticas a implementar en la aplicación se convierte en la meta para alcanzar una arquitectura de trabajo sólida, basada en estándares, escalable y orientada a la implementación de buenas prácticas en el desarrollo.

Para ello se debe realizar un estudio exhaustivo de las herramientas y lenguajes que se adecuan a los requerimientos y presentan un espectro de funciones correspondientes a las necesidades. Dada la política hostil que el gobierno norteamericano mantiene sobre Cuba, el bloqueo económico que prohíbe el acceso a las nuevas tecnologías de la informática y las comunicaciones, la premisa en la selección de las herramientas y tecnologías de soporte al desarrollo fue su clasificación como software libre. Las razones para utilizar software libre son múltiples y variadas.

En los puntos que siguen a continuación se exponen de una manera clara los más importantes:

- Está comprobado que es software de mayor calidad que el propietario (dispone del mayor soporte técnico del mundo).
- En software libre no hay situaciones “lock in” o de dependencia. Al ser el código libre nunca se depende de una empresa que lo posea, posibilitando en cualquier momento rescindir un contrato con la misma sin miedo a problemas en la gestión de la empresa o a situaciones similares.
- Por seguridad informática (accesos no deseados, puertas traseras, funcionalidades no documentadas). Está demostrado que el software libre es más seguro que el propietario, y eso sin tener en cuenta los problemas con virus, troyanos,..
- Estándares abiertos, no dependientes de una empresa.
- Documentado en multitud de idiomas. El software libre cuenta con el mayor equipo de traducción de documentación existente.

CAPÍTULO 2

ARQUITECTURA DEL SISTEMA

El comité de arquitectura compuesto por las principales entidades desarrolladores de software Softel y la UCI, especialistas del MINSAP e Infomed como gestor y proveedor de servicios de red y contenidos especializados, ha definido varias combinaciones de software capaces de soportar las aplicaciones producidas para el SNS entre las que se destacan:

- Linux, Apache, MySQL, PHP (LAMP)
- Linux, Apache, PostgreSQL, PHP
- Linux, Tomcat + JVM + JDK, MySQL, Java

Una vez estudiado y analizado las posibles combinaciones compuestas principalmente por: Linux, como sistema operativo, los servidores Web Apache y Tomcat, MySQL y PostgreSQL como servidores de base de datos y finalmente PHP y Java como lenguajes de programación. Se decidió incorporar a la plataforma tecnológica del grupo la tecnología LAMP considerando que es la que más se adapta a las necesidades planteadas por el grupo y por el cliente con aspectos como el SGBD MySQL.

LAMP es el acrónimo de Linux, Apache, MySQL y PHP, consideradas como unas de las mejores herramientas que el software libre puede proporcionar y que permiten a cualquier organización o individuo tener un servidor Web versátil y poderoso, independientemente del hecho que no es necesario pagar licencias por su utilización, su mantenimiento se reduce a actualizar paquetes que se pueden descargar por Internet y su nivel de seguridad es muy bueno, al liberarse parches de seguridad al muy poco tiempo que se declara una alerta. Conforma una potente y robusta plataforma para el desarrollo y distribución de aplicaciones basadas en la Web.

Una característica muy interesante es el hecho que estos cuatro productos pueden funcionar en una amplia gama de hardware, con requerimientos relativamente pequeños pero que no por eso dejan de ser menos estables que en equipos de grandes capacidades.

Sistema Operativo

Linux es la base de la tecnología LAMP y soporta todas las demás aplicaciones, provee un conjunto de herramientas propias del sistema operativo permitiendo una mayor estabilidad y a diferencia de otros sistemas operativos, ofrece seguridad, estabilidad y desempeño.

Ubuntu es una distribución GNU/Linux que ofrece un sistema operativo predominantemente enfocado a ordenadores de escritorio aunque también proporciona soporte para servidores.

La filosofía de Ubuntu se basa en los siguientes principios:

- Ubuntu siempre será gratuito, y no habrá un coste adicional para la «edición profesional»; se quiere que lo mejor del trabajo esté libremente disponible para todos.
- Para hacer que Ubuntu pueda ser usado por el mayor número de personas posible, Ubuntu emplea las mejores herramientas de traducción y accesibilidad que la comunidad del software libre es capaz de ofrecer.
- Ubuntu publica de manera regular y predecible, una nueva versión cada seis meses. Puede usar la versión estable o probar y ayudar a mejorar la versión en desarrollo.
- Ubuntu está totalmente comprometido con los principios de desarrollo del software de código abierto, se motiva a las personas a utilizarlo, mejorarlo y compartirlo.

Son innumerables las ventajas y beneficios que posee la utilización de Ubuntu.

- Configurar una red en Ubuntu resulta sencillo. Posee herramientas de redes para configurarla de manera fácil, rápida y sobre todo segura, ya que posee IPTABLES que es el firewall más seguro y viene con el kernel.
- Actualizaciones de seguridad muy seguidas. ¿Eso quiere decir que Ubuntu es poco seguro? Todo lo contrario, Ubuntu es una distribución muy segura y enfocada al entorno del usuario, estas actualizaciones se dan según un programa cambie de versión. En pocas palabras, siempre estarás actualizado con Ubuntu.
- Gestor de Paquetes Gráfico, esta puede ser una de las razones más contundentes.

Servidor Web Apache 2.0.9

El servidor Web Apache provee un mecanismo para ofrecer la aplicación Web a los usuarios. Es estable y tiene la capacidad para soportar aplicaciones críticas. El 65% de los sitios Web son servidos por apache. PHP se integra con apache mediante un módulo, de forma que se puedan servir páginas dinámicas y utilizar Web Services. Apache es el servidor Web más usado en sistemas Linux. Los servidores Web se usan para servir páginas Web solicitadas por equipos cliente. Los clientes normalmente solicitan y muestran páginas Web mediante el uso de navegadores Web como Firefox, Opera o Mozilla.

El servidor usado va a ser Apache, que por su robustez, configurabilidad y estabilidad hacen que cada vez más millones de servidores reiteren su confianza en este programa y su licencia es descendiente de la Berkeley Software Distribution (BSD), que permite la modificación del código fuente. Ha alcanzado gran popularidad en ámbitos empresariales debido, entre otras características a:

- Corre en una multitud de sistemas operativos, lo que lo hace prácticamente universal.
- Constituye una tecnología gratuita de código fuente abierta.
- Apache es un servidor altamente configurable de diseño modular. Es muy sencillo ampliar sus capacidades mediante la integración o la construcción de módulos.
- Permite personalizar la respuesta ante los posibles errores que se puedan dar en el servidor.
- Es posible configurar Apache para que ejecute un determinado script cuando ocurra un error en concreto.

Algunas de las características deseables en un Sistema Gestor de base de datos SGBD son:

- Control de la redundancia: La redundancia de datos tiene varios efectos negativos (duplicar el trabajo al actualizar, desperdicia espacio en disco, puede provocar inconsistencia de datos) aunque a veces es deseable por cuestiones de rendimiento.
- Restricción de los accesos no autorizados: cada usuario ha de tener unos permisos de acceso y autorización.
- Cumplimiento de las restricciones de integridad: el SGBD ha de ofrecer recursos para definir y garantizar el cumplimiento de las restricciones de integridad.

Sistema Gestor de Base de Datos MySQL 5.3.2

Un SGBD es un conjunto de programas que permiten crear y mantener una base de datos, asegurando su integridad, confidencialidad y seguridad. Por tanto debe permitir:

- Definir una base de datos: especificar tipos, estructuras y restricciones de datos.
- Construir la base de datos: guardar los datos en algún medio controlado por el mismo SGBD
- Manipular la base de datos: realizar consultas, actualizarla, generar informes.
- Así se trata de un software de propósito general. Ejemplo de SGBD son Oracle y SQL Server de Microsoft.

MySQL es un servidor de bases de datos SQL rápido, robusto, multihilos y multiusuario. Está diseñado para su uso en sistemas de producción para misiones críticas y de alta carga, así como para ser encajado dentro de software desarrollado para las masas. Aunque quedó definido como SGBD MySQL por razones anteriormente explicadas se realizó un estudio de otros SGBD por si en futuras versiones el cliente o el equipo de desarrollo deciden emigrar hacia otro sistema por cuestiones razonables.

Principales características:

- Múltiples motores de almacenamiento InnoDB, MyISAM y Clúster.
- Soporte a SSL, Triggers, VARCHAR, Vistas Actualizables.
- Replicación: múltiples maestros a múltiples esclavos y múltiples esclavos a un maestro.

SGBD Oracle

Se considera como unos de los SGBD más completos, destacando su soporte de transacciones, estabilidad, escalabilidad y el ser multiplataforma. Maneja BD relacionales que hacen uso de los recursos del sistema informático en cualquier hardware, para garantizar al máximo su aprovechamiento en ambientes donde se maneje mucha información. Corre automáticamente en decenas de arquitecturas de hardware y de software, sin tener la necesidad de cambiar una sola línea de código, esto se debe a que más del 80% de los códigos internos de Oracle son iguales a los establecidos en las plataformas de sistemas. Su principal desventaja radica en que es un software propietario, además de su elevado precio, lo que hace que solo se vea en grandes empresas y multinacionales presentando un alto costo de soporte técnico.

SGBD PostgreSQL

Constituye un SGBD de alto nivel que ofrece un conjunto de ventajas: el código fuente está disponible para todos sin costo, es un software multiplataforma, facilita la migración de datos procedentes de otros SGBD, brinda gran consistencia y seguridad en los datos. Como inconvenientes se puede mencionar que consume muchos recursos por lo carga más el sistema y el límite del tamaño de cada fila de las tablas a 8k.

Lenguaje de Programación PHP 5.3.2

La utilización de PHP al unísono con el manejador de bases de datos MySQL, permite la implementación de sistemas Web complejos y multiplataforma con costos bajos y un esquema de seguridad muy completo. PHP es un lenguaje de programación simple y eficiente, hecho pensando en la Web. Es la parte que conecta todo el sistema LAMP. Se utiliza para escribir el contenido dinámico de las páginas. Permite manipular la información guardada en las bases de datos de MySQL, así como a la mayoría de las utilidades de Linux e interactuar con otros componentes o servicios Web.

La combinación LAMP incorporada a la plataforma de trabajo del grupo resulta insuficiente por lo que se deben incorporar otros elementos a dicha plataforma para alcanzar las metas trazadas.

Integrated Development Enviroment

Un entorno de desarrollo integrado o, en inglés, Integrated Development Enviroment ('IDE'), es un programa compuesto por un conjunto de herramientas para un programador. 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. Los IDEs pueden ser aplicaciones por sí solas o pueden ser parte de aplicaciones existentes. Entre los IDEs con licencia gratuita estudiados se encuentran Aptana Studio, Eclipse y Zend Studio for Eclipse:

Aptana Studio es un IDE de código libre para aplicaciones de la Web 2.0. Está focalizada en el desarrollo Web en Java, con soporte a HTML, CSS y JavaScript, así como opcionalmente a otras tecnologías como PHP y Ajax. Está disponible como una aplicación independiente o como plugin para Eclipse. Puede distribuirse para todos los sistemas operativos más comunes: Windows, Linux y Mac OS.

Eclipse es un IDE para todo tipo de aplicaciones, inicialmente desarrollado por IBM, y actualmente gestionado por la Fundación Eclipse, una organización independiente sin ánimo de lucro que fomenta una comunidad de código abierto. La característica clave de Eclipse es la extensibilidad. Eclipse es una gran estructura formada por un núcleo y muchos plugins que van conformando la funcionalidad final. La forma en que los plugins interactúan es mediante interfaces o puntos de extensión. Los lenguajes con proyectos más importantes son: Java, PHP y C/C++. Además existen paquetes que ayudan también con lenguajes para Web como HTML y CSS.

Zend Studio for Eclipse

Se trata de un programa de la casa Zend, impulsores de la tecnología de servidor PHP, orientada a desarrollar aplicaciones Web. Consta de dos partes, las funcionalidades de parte del cliente y las del servidor. Ambas partes se instalan por separado, la del cliente contiene el interfaz de edición y la ayuda. Permite además hacer depuraciones simples de scripts, aunque para disfrutar de toda la potencia de la herramienta de depuración habrá que disponer de la parte del servidor, que instala Apache y el módulo PHP. La interfaz está compuesta por un explorador de archivos, los menús, una ventana de depuración, y otra para mostrar el código de las páginas. Su editor permite escribir los scripts, es bastante útil para la programación en PHP. Dentro de la ayuda podemos encontrar funciones como editar varios archivos, moverse fácilmente entre ellos, marcar a qué elementos corresponden los inicios y cierres de las etiquetas, paréntesis o llaves, moverse al principio o al final de una función, identificación automática del código, entre otros. La herramienta de depuración permite ejecutar páginas y conocer en todo momento el contenido de las variables de la aplicación y las variables del entorno como las cookies, las recibidas por formulario o en la sesión. Posibilita colocar puntos de parada de los scripts y realizar las acciones típicas de depuración.

Por el estudio realizado se arribó a la conclusión de elegir como IDE Zend Studio for Eclipse por ser el que más se adapta a las necesidades del grupo, por corresponderse con la arquitectura y todas las prestaciones que brinda el mismo.

Framework

Es un subsistema de software parcialmente construido, de propósito general para un tipo específico de problema, el cual debe ser instanciado para resolver un problema particular. Típicamente un framework se construye a partir de patrones de diseño. El framework impone patrones de diseño para su uso. Define la arquitectura para una familia de subsistemas. Provee bloques básicos de construcción y adaptadores.

Un framework, en el desarrollo de software, es una estructura de soporte definida mediante la cual otro proyecto de software puede ser organizado y desarrollado. Típicamente, puede incluir soporte de programas, bibliotecas y un lenguaje interpretado entre otros software para ayudar a desarrollar y unir los diferentes componentes de un proyecto. Representa una arquitectura de software que modela las relaciones generales de las entidades del dominio. Provee una estructura y una metodología de trabajo la cual extiende o utiliza las aplicaciones del dominio.

Un framework simplifica el desarrollo de una aplicación mediante la automatización de algunos de los patrones utilizados para resolver las tareas comunes. Además, un framework proporciona estructura al código fuente, forzando al desarrollador a crear código más legible y más fácil de mantener. Por último, un framework facilita la programación de aplicaciones, ya que encapsula operaciones complejas en instrucciones sencillas. (12)

Para las aplicaciones desarrolladas en PHP existe una cantidad considerable de framework que pueden ser utilizados. Entre los que son libres se pueden encontrar a Zend Framework, Kumbia y Symfony.

Zend Framework (ZF) es un marco de trabajo de código abierto orientado a objetos, implementado en PHP 5 y bajo la licencia BSD. Tiene como objetivo simplificar el desarrollo Web encapsulando al mismo tiempo las mejores prácticas de programación en PHP, y provee componentes para el patrón MVC. Entre los componentes que proporciona se encuentran aquellos que se utilizan con frecuencia en aplicaciones Web, incluyendo la autenticación y autorización a través de listas de control de acceso (ACL), la configuración de aplicación, los datos de la memoria caché, el filtrado y validación de los datos proporcionados por el usuario, la internacionalización, las interfaces para Ajax, así como la composición y entrega de correo electrónico.

Kumbia es un framework para el desarrollo de aplicaciones Web escrito en PHP5. Su principal objetivo es fomentar la velocidad y eficiencia en la creación y mantenimiento de aplicaciones Web. Ha sido probado en software comercial y educativo con variedad de demanda y funcionalidad. Este framework está basado en los conceptos de compatibilidad y flexibilidad; además su instalación, configuración y aprendizaje son muy fáciles de realizar; brindando soporte a prácticas y patrones de programación productivos y eficientes.

Symfony es un enorme conjunto de herramientas y utilidades que simplifican el desarrollo de las aplicaciones Web. Emplea el tradicional patrón MVC para separar las distintas partes que forman una aplicación Web. La arquitectura MVC proporciona grandes ventajas, como la organización del código, la reutilización, la flexibilidad y una programación mucho más entretenida.

Por si fuera poco, crear la aplicación con Symfony permite crear páginas XHTML válidas, depurar fácilmente las aplicaciones, crear una configuración sencilla, abstracción de la base de datos utilizada, enrutamiento con URL limpias, varios entornos de desarrollo y muchas otras utilidades para el desarrollo de aplicaciones. Permite olvidarse de las tareas aburridas y repetitivas para centrarse en el trabajo que realmente aporta valor a las aplicaciones.

Symfony proporciona una estructura en forma de árbol de archivos para organizar de forma lógica todos esos contenidos, además de ser consistente con la arquitectura MVC utilizada y con la agrupación proyecto / aplicación / módulo. Cada vez que se crea un nuevo proyecto, aplicación o módulo, se genera de forma automática la parte correspondiente de esa estructura. Además, la estructura se puede personalizar completamente, para reorganizar los archivos y directorios o para cumplir con las exigencias de organización de un cliente.

Symfony es un framework de tipo MVC escrito en PHP 5. Su estructura interna se ha diseñado para obtener lo mejor del patrón MVC y la mayor facilidad de uso. Gracias a su versatilidad y sus posibilidades de configuración constituye un framework adecuado para cualquier proyecto de aplicación Web. Es compatible con la mayoría de gestores de bases de datos, como MySQL, PostgreSQL, Oracle y SQL Server de Microsoft. Se puede ejecutar tanto en plataformas *nix (Unix, Linux, etc.) como en plataformas Windows.

Características

Symfony se diseñó para que se ajustara a los siguientes requisitos:

- Fácil de instalar y configurar en la mayoría de plataformas (y con la garantía de que funciona correctamente en los sistemas Windows y *nix estándares).
- Independiente del sistema gestor de bases de datos.
- Sencillo de usar en la mayoría de casos, pero lo suficientemente flexible como para adaptarse a los casos más complejos.
- Sigue la mayoría de mejores prácticas y patrones de diseño para la Web.
- Preparado para aplicaciones empresariales y adaptables a las políticas y arquitecturas propias de cada empresa, además de ser lo suficientemente estable como para desarrollar aplicaciones a largo plazo.

Framework de Presentación ExtJS

ExtJS es un framework JavaScript del lado del cliente para el desarrollo de aplicaciones Web. Este framework puede correr en cualquier plataforma que pueda procesar POST y devolver datos estructurados (PHP, Java, .NET y algunas otras).

Ventajas de EXT JS

- Código reutilizable
- Independiente o adaptable a framework diferentes (prototype, jquery, YUI)
- Orientada a la programación de interfaces tipo desktop en la Web
- El API es homogenizado independientemente del adaptador usado. Los controles siempre se verán igual.
- Soporte comercial
- Extensa comunidad de usuarios.

ExtJS presenta un framework de desarrollo enteramente en JavaScript que permite la implementación de interfaces visuales muy potentes, ofreciendo componentes para la implementación de tablas, árboles, formularios, contenedores, etc.

La característica de implementación que la marca es el uso intensivo de la librería ExtJS integrado a la tecnología AJAX; con ellas se logra una experiencia de usuario muy parecida o igual a la que se tiene en las aplicaciones de escritorio evitándose las tediosas recargas de páginas completas al solicitarse nuevos contenidos, actividad permanente debido a las especificaciones del producto.
El uso de esta tecnología cliente permite además establecer un balance con el servidor permitiéndole a este atender mayor cantidad de solicitudes. Sin contar el gran aporte estético con el uso de los componentes predefinidos.

Mapeo de Objetos a Bases de datos -- Object Relational Model (ORM)

Las bases de datos siguen una estructura relacional. PHP 5 y Symfony por el contrario son orientados a objetos. Por este motivo, para acceder a la base de datos como si fuera orientada a objetos, es necesaria una interfaz que traduzca la lógica de los objetos a la lógica relacional. Esta interfaz se denomina “mapeo de objetos a bases de datos” (ORM, de sus siglas en inglés “object-relational mapping”). El componente que se encarga por defecto de gestionar el modelo en Symfony es una capa de tipo ORM realizada mediante el proyecto Propel.

En las aplicaciones Symfony, el acceso y la modificación de los datos almacenados en la base de datos se realiza mediante objetos; de esta forma nunca se accede de forma explícita a la base de datos. Este comportamiento permite un alto nivel de abstracción y permite una fácil portabilidad. La principal ventaja que aporta el ORM es la reutilización, permitiendo llamar a los métodos de un objeto de datos desde varias partes de la aplicación e incluso desde diferentes aplicaciones. Un ORM consiste en una serie de objetos que permiten acceder a los datos y que contienen en su interior cierta lógica de negocio. La principal ventaja de la capa de abstracción es la portabilidad, porque hace posible el cambiar la aplicación a otra base de datos, incluso en mitad del desarrollo de un proyecto. Si se debe desarrollar rápidamente un prototipo de una aplicación y el cliente no ha decidido todavía la base de datos que mejor se ajusta a sus necesidades, se puede construir la aplicación utilizando SQL y cuando el cliente haya tomado la decisión, cambiar fácilmente a MySQL, PostgreSQL u Oracle. Solamente es necesario cambiar una línea en un archivo de configuración y todo funciona correctamente.

Propel es un ORM, lo que significa que, gracias a Propel, se puede trabajar con una base de datos sin utilizar instrucciones SQL. Para crear, obtener, modificar o borrar datos de la base de datos, no es necesario escribir ni una sola sentencia SQL, ya que se puede trabajar directamente con objetos. Propel es el ORM clásico de Symfony. Su principal ventaja es que está completamente integrado con Symfony y que decenas de plugins sólo funcionan para Propel. Independientemente de que el ORM definido para ser utilizado en el proyecto es Propel, es válido resaltar que Doctrine es el ORM del futuro de Symfony. Su principal ventaja es el rendimiento en ejecución y la forma tan concisa en la que se pueden escribir consultas muy complejas. Su principal problema es que todavía no dispone del mismo nivel de integración que Propel.

Al emplear las capas de abstracción de objetos/relacional evita utilizar una sintaxis específica de un sistema de bases de datos concreto. Esta capa transforma automáticamente las llamadas a los objetos en consultas SQL optimizadas para el sistema gestor de bases de datos que se está utilizando en cada momento. De esta forma, es muy sencillo cambiar a otro sistema de bases de datos completamente diferente en mitad del desarrollo de un proyecto. Estas técnicas son útiles por ejemplo cuando se debe desarrollar un prototipo rápido de una aplicación y el cliente aun no ha decidido el sistema de bases de datos que más le conviene.

El prototipo se puede realizar utilizando SQLite y después se puede cambiar fácilmente a MySQL, PostgreSQL u Oracle cuando el cliente se haya decidido. El cambio se puede realizar modificando solamente una línea en un archivo de configuración.

La capa de abstracción utilizada encapsula toda la lógica de los datos. El resto de la aplicación no tiene que preocuparse por las consultas SQL y el código SQL que se encarga del acceso a la base de datos es fácil de encontrar. Los desarrolladores especializados en la programación con bases de datos pueden localizar fácilmente el código.

Herramienta para el Diseño Visual de Bases de Datos

DB Designer Fork, es un programa de diseño visual de bases de datos que integra el diseño entidad-relación, modelado, la creación de bases de datos y mantenimiento de estas en un único entorno. Una vez creada la base de datos, se podrá crear los scripts correspondientes para diferentes gestores, como Firebird/InterBase, Microsoft SQL Server, MySQL, Oracle o PostgreSQL.

Esta herramienta está disponible para diferentes sistemas operativos, lo cual se le atribuye la capacidad de ser multiplataforma. Además posibilita la generación del esquema de la base de datos definida por el usuario, permite la sincronización del modelo con la base de datos, soporta índices y todos los tipos de campos de MySQL, así como todos los tipos de campos que defina el usuario.

Gestión de Proyecto-Gestión Documental-Control y seguimiento de errores

DotProject es una herramienta para la gestión de proyectos basada en Web, está escrita en PHP y MySQL, incluye módulos para compañías, tareas (con gráficas Gantt), foros, permite upload de archivos, calendarios, soporte multilenguaje, entre otras muchas características que lo hacen una herramienta muy completa. DotProject fue creado por dotmarketing.org en el año 2000, con el fin de construir una herramienta para la Gestión de Proyectos. Está construido por aplicaciones de código abierto y es mantenida por un dedicado grupo de voluntarios. Es una aplicación basada en Web, multiusuario, soporta varios lenguajes y es Software Libre.

Está programado en PHP, y utiliza inicialmente MySQL como base de datos (aunque otros motores como PostgreSQL también pueden ser utilizados). La plataforma recomendada para utilizar DotProject se denomina LAMP (Linux + Apache + MySQL + PHP). El grupo que desarrolla DotProject se basa en los siguientes puntos:

- Proveer a los usuarios de funcionalidad orientada a la Gestión de Proyectos.
- Construir una herramienta con una interfaz de usuario simple, claro y consistente.
- Ser de código abierto, libre acceso y utilización.

Requisitos

- Servidor Web (se recomienda Apache 1.3.27 o superior)
- PHP 4.1.x o superior
- MySQL 3.23.51 o superior

Metodología de desarrollo de software

Un proceso de desarrollo de software detallado y completo suele denominarse "metodología". Las metodologías se basan en una combinación de los modelos de proceso genéricos (cascada, iterativo, evolutivo, incremental). Adicionalmente una metodología debería definir con precisión los artefactos, roles y actividades involucrados, junto con prácticas y técnicas recomendadas, guías de adaptación de la metodología al proyecto, guías para uso de herramientas de apoyo, etc. Las metodologías de desarrollo de software especifican cómo se debe dividir un proyecto en etapas, qué tareas se llevan a cabo en cada etapa, qué salidas se producen y cuándo se deben producir, qué restricciones se aplican, qué herramientas se van a usar y cómo se gestionará y controlará un proyecto.
Constituyen un conjunto de actividades necesarias para transformar los requisitos de los usuarios en un sistema software. Las metodologías de desarrollo de software se dividen en dos grandes grupos: las metodologías ágiles y tradicionales.

Las metodologías tradicionales son aquellas que están guiadas por una fuerte planificación durante todo el proceso de desarrollo; llamadas también metodologías pesadas, robustas o clásicas, donde se realiza una intensa etapa de análisis y diseño antes de la construcción del sistema, unas de las más empleadas en el mundo de la producción de software son Rational Unified Process (RUP) y Microsoft Solution Framework (MSF).

Las metodologías ágiles por su parte son aquellas cuando el desarrollo de software es incremental (entregas pequeñas de software, con ciclos rápidos), cooperativo (cliente y desarrolladores trabajan juntos constantemente con una cercana comunicación), sencillo (el método en sí mismo es fácil de aprender y modificar, bien documentado), y adaptable (permite realizar cambios de último momento). Las metodologías ágiles más utilizadas en la actualidad dentro del mundo de la producción de software se encuentran: eXtreme Programming (XP), Scrum, Familia de Metodologías Crystal.

- La Metodología RUP es más adaptable para proyectos de largo plazo.
- La Metodología MSF se adapta a proyectos de cualquier dimensión y de cualquier tecnología.
- La Metodología XP en cambio, se recomienda para proyectos de corto plazo.

Extreme Programming (XP)

Es una de las metodologías de desarrollo de software más exitosas en la actualidad utilizada para proyectos de corto plazo, pequeños equipos y cuyo plazo de entrega era ayer. La metodología consiste en una programación rápida o extrema, cuya particularidad es tener como parte del equipo, al usuario final, pues es uno de los requisitos para llegar al éxito del proyecto.

Características de XP, la metodología se basa en:

- Pruebas Unitarias: se basa en las pruebas realizadas a los principales procesos, de tal manera que adelantándonos en algo hacia el futuro, podamos hacer pruebas de las fallas que pudieran ocurrir.
- Refabricación: se basa en la reutilización de código, para lo cual se crean patrones o modelos estándares, siendo más flexible al cambio.
- Programación en pares: una particularidad de esta metodología es que propone la programación en pares, la cual consiste en que dos desarrolladores participen en un proyecto en una misma estación de trabajo. Cada miembro lleva a cabo la acción que el otro no está haciendo en ese momento. Es como el chofer y el copiloto: mientras uno conduce, el otro consulta el mapa.

¿Qué es lo que propone XP?

- Empieza en pequeño y añade funcionalidad con retroalimentación continua
- El manejo del cambio se convierte en parte sustantiva del proceso
- El costo del cambio no depende de la fase o etapa
- No introduce funcionalidades antes que sean necesarias
- El cliente o el usuario se convierten en miembro del equipo

Lo fundamental en este tipo de metodología es:

- La comunicación, entre los usuarios y los desarrolladores
- La simplicidad, al desarrollar y codificar los módulos del sistema
- La retroalimentación, concreta y frecuente del equipo de desarrollo, el cliente y los usuarios finales

Microsoft Solution Framework (MSF)

Esta es una metodología flexible e interrelacionada con una serie de conceptos, modelos y prácticas de uso, que controlan la planificación, el desarrollo y la gestión de proyectos tecnológicos. MSF se centra en los modelos de proceso y de equipo dejando en un segundo plano las elecciones tecnológicas.

MSF tiene las siguientes características:

- Adaptable: es parecido a un compás, usado en cualquier parte como un mapa, del cual su uso es limitado a un específico lugar.
- Escalable: puede organizar equipos tan pequeños entre 3 o 4 personas, así como también, proyectos que requieren 50 personas a más.
- Flexible: es utilizada en el ambiente de desarrollo de cualquier cliente.
- Tecnología Agnóstica: porque puede ser usada para desarrollar soluciones basadas sobre cualquier tecnología.

MSF se compone de varios modelos encargados de planificar las diferentes partes implicadas en el desarrollo de un proyecto: Modelo de Arquitectura del Proyecto, Modelo de Equipo, Modelo de Proceso, Modelo de Gestión del Riesgo, Modelo de Diseño de Proceso y finalmente el modelo de Aplicación.

Rational Unified Process (RUP) Proceso Unificado de Desarrollo de Software

Aunque en el caso particular de RUP, por el especial énfasis que presenta en cuanto a su adaptación a las condiciones del proyecto (mediante su configuración previa a aplicarse), realizando una configuración adecuada, podría considerarse Ágil.

La metodología RUP, llamada así por sus siglas en inglés Rational Unified Process, divide en 4 fases el desarrollo del software:

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

Cada una de estas etapas es desarrollada mediante el ciclo de iteraciones, la cual consiste en reproducir el ciclo de vida en cascada a menor escala. Los objetivos de una iteración se establecen en función de la evaluación de las iteraciones precedentes. Vale mencionar que el ciclo de vida que se desarrolla por cada iteración, es llevada bajo dos disciplinas:

Disciplina de Desarrollo

- Modelamiento del Negocio: Entendiendo las necesidades del negocio.
- Requerimientos: Trasladando las necesidades del negocio a un sistema automatizado.
- Análisis y Diseño: Trasladando los requerimientos dentro de la arquitectura de software.
- Implementación: Creando software que se ajuste a la arquitectura y que tenga el comportamiento deseado.
- Pruebas: Asegurándose que el comportamiento requerido es el correcto y que todo lo solicitado está presente.
- Despliegue e Instalación: Hacer todo lo necesario para la salida del proyecto

Disciplina de Soporte

- Configuración y administración del cambio: Guardando todas las versiones del proyecto.
- Administrando el proyecto: Administrando horarios y recursos.
- Ambiente: Administrando el ambiente de desarrollo.

Los elementos del RUP son:

- Actividades: Son los procesos que se llegan a determinar en cada iteración.
- Trabajadores: Vienen hacer las personas o entes involucrados en cada proceso.
- Artefactos: Un artefacto puede ser un documento, un modelo, o un elemento de modelo.

El proceso de decidir qué metodología implementar es un trabajo constante, y debe responder a las necesidades del negocio y al crecimiento del equipo. Cada proyecto de software requiere de una forma de particular de abordar el problema. Independientemente de que se han analizados varias de las metodologías más utilizadas en el mundo de la producción de software, cada una con sus ventajas y particularidades, se determinó elegir la metodología RUP porque es la que más se adapta a las necesidades del equipo de acuerdo a sus características, además de que es la metodología que se emplea en el polo de GIB al cual pertenece el grupo de desarrollo.

Lenguaje de modelado

La decisión de utilizar UML, independientemente de que es el lenguaje seleccionado para la descripción de la arquitectura, como notación para el método software se debe a que se ha convertido en un estándar que tiene las siguientes características:

- Permite modelar sistemas utilizando técnicas orientadas a objetos (OO).
- Permite especificar todas las decisiones de análisis, diseño e implementación, construyéndose así modelos precisos, no ambiguos y completos.
- Puede conectarse con lenguajes de programación (Ingeniería directa e inversa).
- Permite documentar todos los artefactos de un proceso de desarrollo (requisitos, arquitectura, pruebas, versiones, etc.).
- Cubre las cuestiones relacionadas con el tamaño propio de los sistemas complejos y críticos.
- Es un lenguaje muy expresivo que cubre todas las vistas necesarias para desarrollar y luego desplegar los sistemas.
- Existe un equilibrio entre expresividad y simplicidad, pues no es difícil de aprender ni de utilizar.
- UML es independiente del proceso, aunque para utilizarlo óptimamente se debería usar en un proceso que fuese dirigido por los casos de uso, centrado en la arquitectura, iterativo e incremental.

Impulsados por la necesidad de métodos más ágiles de desarrollo de software, UML 1.x evoluciona en UML 2.0, siendo dicha evolución influenciada por el Desarrollo Conducido por Modelo (MDD – Model Driven Development) y el Modelamiento Conducido por Arquitectura (MDA – Model Driven Architecture).

UML 2.0 es la mayor revisión que se le ha hecho a UML desde la versión 1.0. El modelo conceptual ha sido reestructurado completamente y nuevos diagramas han sido incorporados. Los diagramas tradicionales también han sido mejorados. La nueva versión permitirá a los fabricantes de herramientas CASE proporcionar a los analistas, arquitectos y desarrolladores; herramientas cada vez más potentes que les permitan aprovechar mejor los modelos y como consecuencia generar una mayor cantidad código reduciendo significativamente el ciclo de desarrollo de sus aplicaciones. (1)

Herramienta de Modelado

Existen herramientas creadas para el desarrollo de la ingeniería de software una de ellas son las herramientas CASE (Computer Aided Software Engineering), que constituyen un conjunto de aplicaciones informáticas destinadas a aumentar la productividad en el desarrollo de software. Las herramientas CASE se acoplan con las metodologías para dar una forma de representar sistemas y suponen una forma de abstracción del código fuente, a un nivel donde la arquitectura y el diseño se hacen más aparentes y fáciles de entender y modificar. Cuanto mayor es un proyecto, más importante es el uso de tecnología CASE. Entre las herramientas CASE orientadas a UML están: ArgoUML, Poseidon for UML, MagicDraw UML, Visual Paradigm y Borland Together. Entre las herramientas para el modelado más utilizadas tanto en la docencia como en la producción en la UCI resaltan: Rational Rose, Visual Paradigm y Enterprise Architect. (13)

Rational Rose Enterprise Edition.

Es la herramienta que permite construir y desarrollar a través del UML los casos de uso diferentes diagramas, modelando los flujos de trabajo por los que transita el desarrollo de un software. Permite la realización de los diferentes diagramas y la posterior generación del código, todo orientado a objetos. Posee librerías que facilitan la obtención de una ingeniería inversa sobre diferentes lenguajes como Java, C++, Corba, XML_DTD, ADA, Visual Basic. Esta herramienta posibilita gestionar la evolución del ciclo de vida de un proyecto de software. Racional Rose aligera la implementación al automatizar los modelos arquitectónicos. Además permite visualizar, entender, y refinar los requerimientos y arquitectura antes de introducirse en el código. Establece una plataforma para automatizar arquitecturas de mejores prácticas hechas para soluciones tecnológicas específicas.

Rational Rose presenta características que todos los desarrolladores, analistas, y arquitectos exigen como son: un soporte UML incomparable, completo soporte al equipo, desarrollo basado en componentes, etc. La Suite del Rational brinda una serie de productos que complementan su funcionamiento a la hora de implementar un software, ofreciendo diversas ventajas, como la administración de la configuración del software o de las solicitudes de cambios permitiendo que los administradores o jefes de proyecto puedan llevar un control sobre la evolución del proyecto de software que se implemente.

Enterprise Architect

Enterprise Architect combina el poder de la última especificación UML 2.1 con alto rendimiento, interfaz intuitiva, para traer modelado avanzado al escritorio, y para el equipo completo de desarrollo e implementación. Provee la trazabilidad completa desde el análisis de requerimientos hasta los artefactos de análisis y diseño, a través de la implementación y el despliegue. Provee una generación poderosa de documentos y herramientas de reporte con un editor de plantilla completo. Soporta generación e ingeniería inversa de código fuente para muchos lenguajes populares, incluyendo C++, C#, Java, Delphi, VB.Net, Visual Basic y PHP.

Visual Paradigm for UML Community Edition.

Visual Paradigm es una herramienta CASE para modelamiento UML muy potente, gratuita, fácil de instalar, utilizar y actualizar. Te permite dibujar todo tipo de diagramas UML, revertir código fuente a modelos UML, generar código fuente desde los diagramas UML, y mucho más. Incluye los objetos más recientes de UML además de diagramas de casos de uso, diagramas de clase, diagramas de componentes, reversa instantánea para Java, C++, DotNet Exe/dll, XML, XML Schema, y Corba IDL, ofrece soporte para Rational Rose, integración con Microsoft Visio, además permite generar reportes y documentación en HTML/PDF.

Visual Paradigm es una herramienta UML profesional que soporta el ciclo de vida completo del desarrollo de software: análisis y diseño orientados a objetos, construcción, pruebas y despliegue. El software de modelado UML ayuda a una más rápida construcción de aplicaciones de calidad, mejores y a un menor coste. Permite dibujar todos los tipos de diagramas de clases, código inverso, generar código desde diagramas y generar documentación. La herramienta UML CASE también proporciona abundantes tutoriales de UML, demostraciones interactivas de UML y proyectos UML.

Luego de estudiadas y analizadas las características de las herramientas informáticas anteriores, se selecciona como herramienta CASE para dar soporte al desarrollo del sistema a Visual Paradigm Suite for UML 6.4

Sistema de Salvas Automáticas

Bacula es fácil de usar y muy eficiente, ofrece importantes características avanzadas de almacenamiento de datos que hacen posible encontrar y recuperar muchos archivos dañados. Bacula está conformado por varios servicios importantes: director de la consola, archivo, almacenamiento, y servicios de monitoreo. Bacula es un conjunto de paquetes informáticos que permiten administrar los respaldos, recuperaciones y verificaciones de los datos de una máquina a través de una red de diferentes computadoras. Puede correr sobre una sola computadora y puede respaldar hacia varios tipos de medio de almacenamiento, incluyendo casetes y discos duros. En general pude decirse que BACULA es un programa de respaldos en red basado en la filosofía Cliente/Servidor. (14)

Fue diseñado para sistemas BSD, Linux, Mac, OS X, Unix y Microsoft. Por otro lado su arquitectura se basa fundamentalmente en una colección de demonios o servicios que se ejecutan en segundo plano (background) y cooperan entre sí para realizar copias de respaldo de los archivos necesarios.

Conocidas algunas de las características, sería importante conocer algunos conceptos de palabras tratadas anteriormente para luego continuar profundizando en el sistema BACULA. Fundamentalmente respaldo es la obtención de una copia de datos en otro medio magnético, de tal modo que a partir de dicha copia es posible restaurar el sistema al momento de haber realizado el respaldo. Por otro lado recuperación es la tarea que se lleva a cabo cuando es necesario volver al estado de la aplicación al momento del último respaldo.

De estas dos definiciones puede concluirse en la importancia de realizar salvas a los datos de un ordenador que puede resumirse en restaurar un grupo de ficheros que hayan sido borrados o dañados de la computadora, aunque también pueden realizarse respaldos del sistema completo para si en algún momento se desea regresar a un estado operacional previo a un problema en el sistema operativo, finalmente se propone utilizar en el grupo la versión Bacula 2.4.2.

Sistema de control de versiones

Históricamente el servidor de control de versiones "Concurrent Versions System" (CVS) ha sido el sistema más utilizado pero dado las limitaciones de CVS, Subversion a tomado relevo como un sustituto válido

Subversion es un sistema de control de versiones open source. Mediante Subversion, puede registrar la historia de archivos de código fuente y documentos. Gestiona archivos y directorios a través del tiempo. Se coloca un árbol de archivos en un repositorio central. El repositorio es como un servidor de archivos ordinario, excepto en que recuerda todos los cambios que se realizan sobre los archivos y los directorios que contiene.

Para acceder a un repositorio de Subversion mediante el protocolo HTTP, se debe instalar y configurar un servidor Web. Apache 2 ha demostrado funcionar bien con Subversion. Para acceder al repositorio de Subversion mediante el protocolo HTTPS, se debe instalar y configurar un certificado digital en tu servidor Web Apache 2.

Diferencias entre CVS y Subversión. Las principales ventajas que le ofrece Subversion sobre CVS:

- Fuerte integración con Apache: Esto permite definir controles de acceso avanzados y navegación vía Web para consultar el depósito de archivos, proceso carente en CVS.
- Transparencia al eliminar y cambiar nombres de archivos: Si ha intentando este último proceso en CVS, seguramente sabrá que requiere intervención manual en el depósito para lograrlo, Subversion contempla esta deficiencia y la corrige con éxito.
- Copias ligeras sobre ramificaciones: La generación de ramificaciones en CVS además de ser un proceso involucrado implica la generación de una copia nueva en el depósito, mismo mecanismo que hace crecer exponencialmente el tamaño del depósito, Subversion independientemente del número de ramificaciones creadas mantiene un árbol diferencial de cambios, minimizando así el espacio consumido en el depósito.
- Copias diferenciales de archivos binarios : Basado en el mismo principio de copias ligeras, Subversion es capaz de mantener un control diferencial sobre cualquier archivo binario del depósito así reduciendo el consumo de espacio, esto contrastado con CVS que requiere archivar copias completas de un archivo binario cada vez que éste cambia.

2.4 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.

Las vistas de la arquitectura constituyen un extracto de los modelos que están en la línea base de la arquitectura y se representa a través de 4+1 vistas arquitectónicas propuestas por Phelipe Kruchten [1995] e incorporadas a RUP. Una vista es la representación de un conjunto coherente de elementos arquitectónicos y sus relaciones, en este sentido la vista describe parte de la arquitectura del sistema.

Las vistas de casos de uso y diseño son generalmente necesarias en cualquier sistema, mientras que las tres restantes dependen de la complejidad de su arquitectura. Todas las vistas pueden ser modeladas dinámicamente mediante los diagramas UML de comportamiento, es decir, los diagramas de secuencia, colaboración y actividad. Como se ha comentado antes, estas decisiones y diagramas deben incluirse, argumentarse y relacionarse en el documento de descripción de la arquitectura. La esencia 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. [Ver artefacto Documento de Descripción de la arquitectura en el expediente del proyecto]

Vista de Caso de Uso (Rectora)

La confección del diagrama de casos de uso del sistema 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, diseñada e implementada, facilitándole el trabajo a los propios trabajadores del grupo. La vista en cuestión contiene los casos de usos y los escenarios que abarcan el comportamiento más importante o que representa un mayor riesgo debido a su complejidad. Es representada por un diagrama de casos de uso que es a su vez un subconjunto del modelo de caso de uso del sistema, se obtiene cuando se realiza el flujo de trabajo levantamiento de requisitos de la fase de inicio de la metodología seleccionada. La vista rectora representa los casos de uso significativos para la arquitectura ya que describen alguna funcionalidad importante y crítica o algún requisito que deba priorizarse.

Para priorizar los casos de uso del sistema, es necesario determinar cuáles son necesarios para el desarrollo en las primeras iteraciones y cuáles pueden dejarse para más tarde. Los resultados se recogen en la presente vista de la arquitectura del modelo de casos de uso. Para poder determinar qué caso de uso se desarrollarán en cada release de construcción, RUP propone clasificarlos de acuerdo al impacto que tienen en la arquitectura en críticos, secundarios, auxiliares y opcionales.

Críticos: Constituyen los casos de uso más importantes para los usuarios, porque cubren las principales tareas o funciones que el sistema ha de realizar, definen la arquitectura básica.

Secundarios: Sirven de apoyo a los casos de uso críticos, involucran funciones secundarias y tienen un impacto más modesto sobre la arquitectura, pero deben implementarse pronto porque responden a los requerimientos de interés para los usuarios.

Auxiliares: No son claves para la arquitectura y complementan casos de usos críticos y secundarios.

Opcionales: Responden a funcionalidades que pueden o no estar en la aplicación, pero que no son imprescindibles en las primeras versiones.

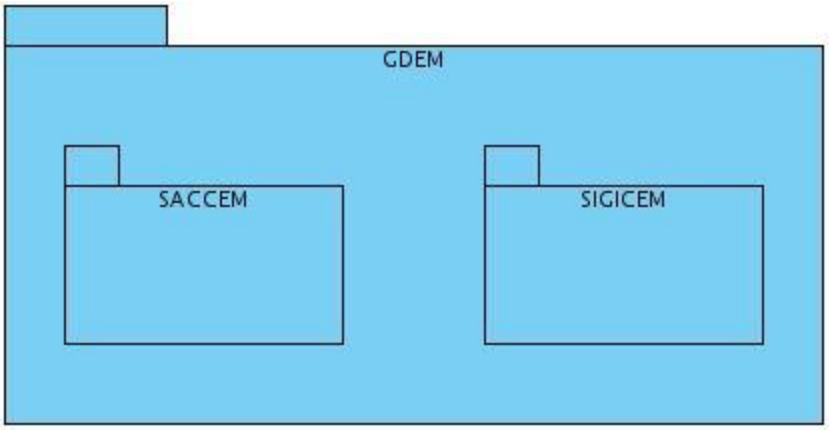


Fig. 3 Organización del GDEM

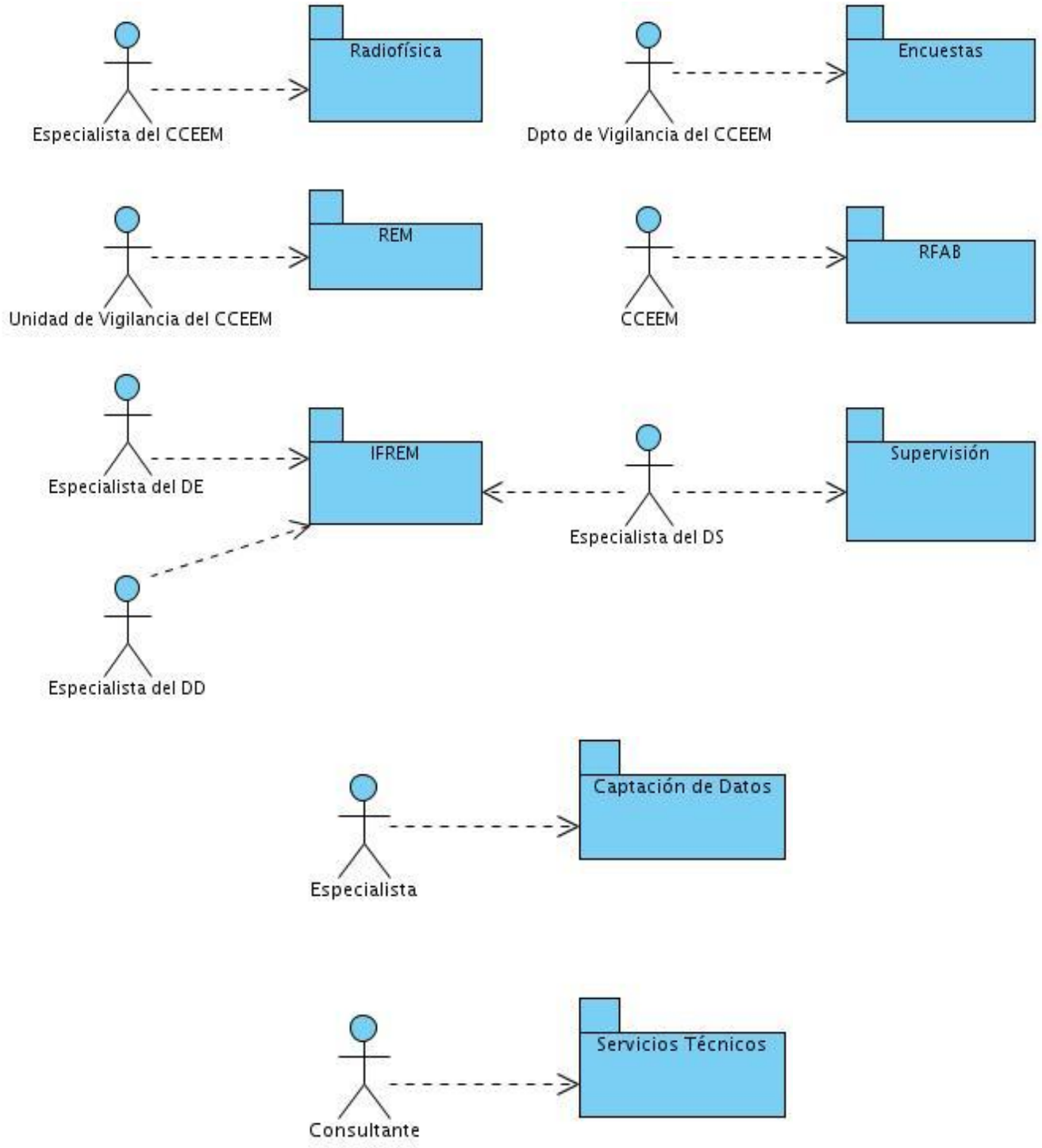


Fig. 4 Organización por paquetes del GDEM

Vista Lógica

La vista lógica contiene las clases del diseño más importantes, organizadas por paquetes y subsistemas en capas de trabajo. Es representada por uno o varios diagramas de clases que son un subconjunto del modelo de diseño. Se obtiene durante el flujo de trabajo de Análisis y Diseño que propone RUP.

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 tendremos la necesidad de organizar los elementos de un diagrama en un grupo. Tal vez se quiera mostrar ciertas clases o componentes, son parte de un subsistema en particular. Para ello se agrupará en un paquete, que se representa por una carpeta tabular. Básicamente un paquete agrupa los elementos de un diagrama.

Symfony toma lo mejor de la arquitectura MVC y la implementa de forma que el desarrollo de aplicaciones sea rápido y sencillo. Implementa el patrón MVC seleccionado para el sistema, por cuanto la estructura organizacional del GDEM se representa a través de tres elementos fundamentales: el modelo, la vista y el controlador.

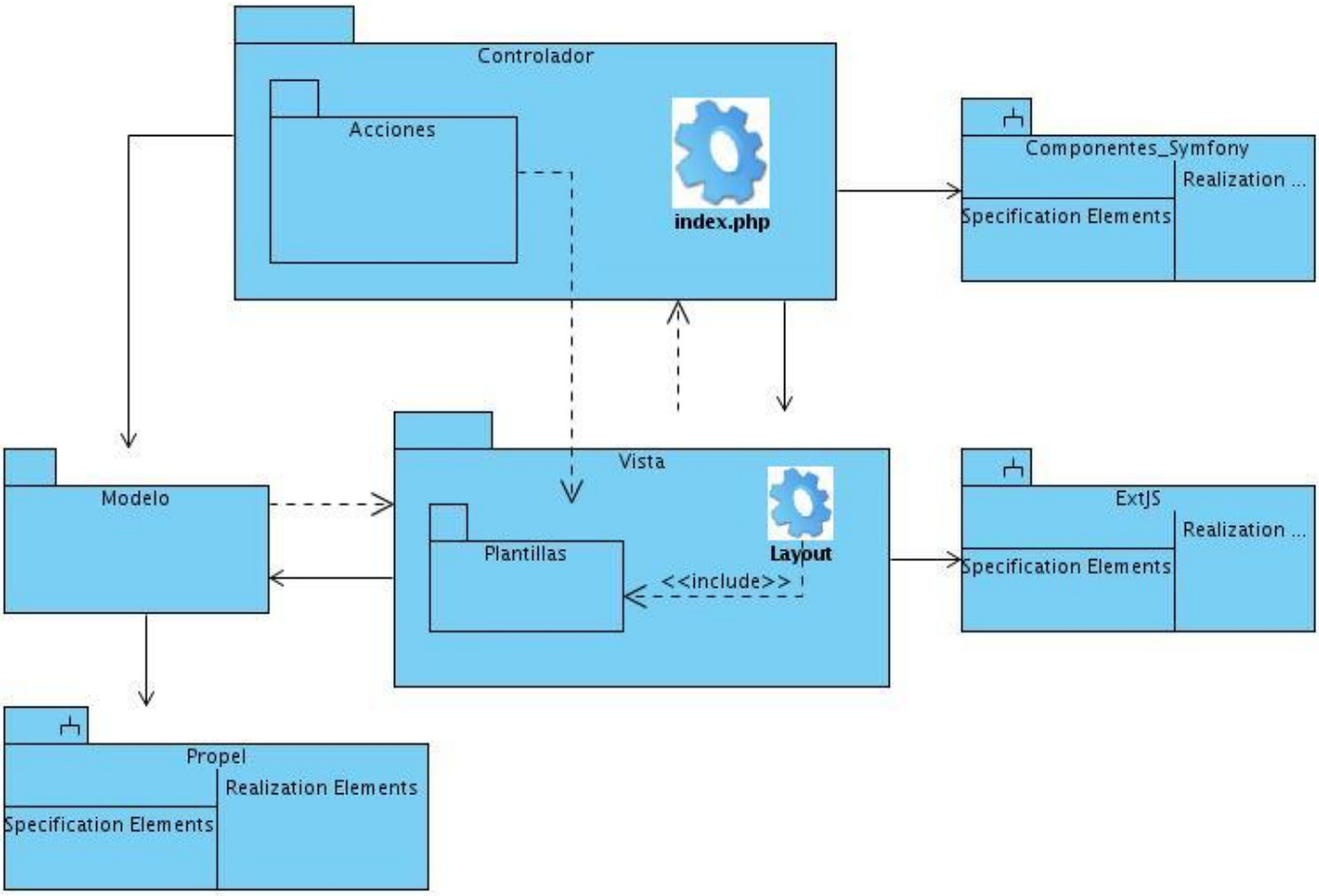


Fig. 5 Vista Lógica General

A continuación se presenta la descripción de los paquetes y subsistemas del diseño que conforman la vista lógica.

La capa del Modelo

- Abstracción de la base de datos
- Acceso a los datos

Paquete del modelo

Contiene las clases que encapsulan la lógica del dominio, y se encargan del acceso a los datos almacenados en el gestor de base de datos. Recoge el código para la manipulación de los datos

Subsistema Propel

Las bases de datos son relacionales. PHP 5 y Symfony están orientados a objetos. Para acceder de forma efectiva a la base de datos desde un contexto orientado a objetos, es necesaria una interfaz que traduzca la lógica de objetos a la lógica relacional. Dicha interfaz se llama ORM. Symfony usa Propel como ORM y a su vez Propel usa Creole como una capa de abstracción.

Propel es un ORM cuya función es gestionar el acceso a la base de datos y gestionar el modelo del sistema. Implica que el acceso y la modificación de los datos almacenados en la base de datos se realicen mediante objetos, nunca de forma explícita, permitiendo un alto nivel de abstracción y fácil portabilidad. Propel tiene incluido tareas para generar automáticamente las sentencias SQL necesarias para crear las tablas de la base de datos. La librería Propel se encarga de esta generación automática, ya que crea el esqueleto o estructura básica de las clases y genera automáticamente el código necesario. Cuando Propel encuentra restricciones de claves foráneas (o externas) o cuando encuentra datos de tipo fecha, crea métodos especiales para acceder y modificar esos datos. La abstracción de la base de datos es completamente transparente para el programador, ya que se realiza de forma nativa mediante PDO (PHP Data Objects). Así, si se cambia el sistema gestor de bases de datos en cualquier momento, no se debe reescribir ni una línea de código, ya que tan sólo es necesario modificar un parámetro en un archivo de configuración.

Symfony automatiza la creación del modelo utilizando la descripción del esquema relacional de la base de datos. Por cada tabla del esquema relacional crea cuatro clases fundamentales: BaseNombreTabla, BaseNombreTablaPeer, NombreTabla, NombreTablaPeer. Dos de ellas con nombre Base y abstractas son generadas directamente a partir del esquema y contienen implementado los métodos básicos de acceso a los datos. Nunca se deberían modificar porque cada vez que se genera el modelo se borrarán y crearán nuevamente. Las otras dos clases de objetos propias heredan de las clases con nombre Base correspondiente. Estas clases no se modifican cuando se actualiza el modelo, por lo que son las clases en las que se añaden los métodos propios. Entre estas últimas se distinguen la de nombre terminado en Peer, lo que significa que contiene métodos estáticos para trabajar sobre las tablas de la base de datos, proporcionando los medios necesarios para obtener los registros de las tablas. Sus métodos devuelven normalmente un objeto o una colección de objetos de la clase NombreTabla correspondiente. La clase NombreTabla es por su parte la encargada de representar un registro de la base de datos. El Paquete Modelo contiene todas las clases necesarias para el acceso a los datos del sistema.

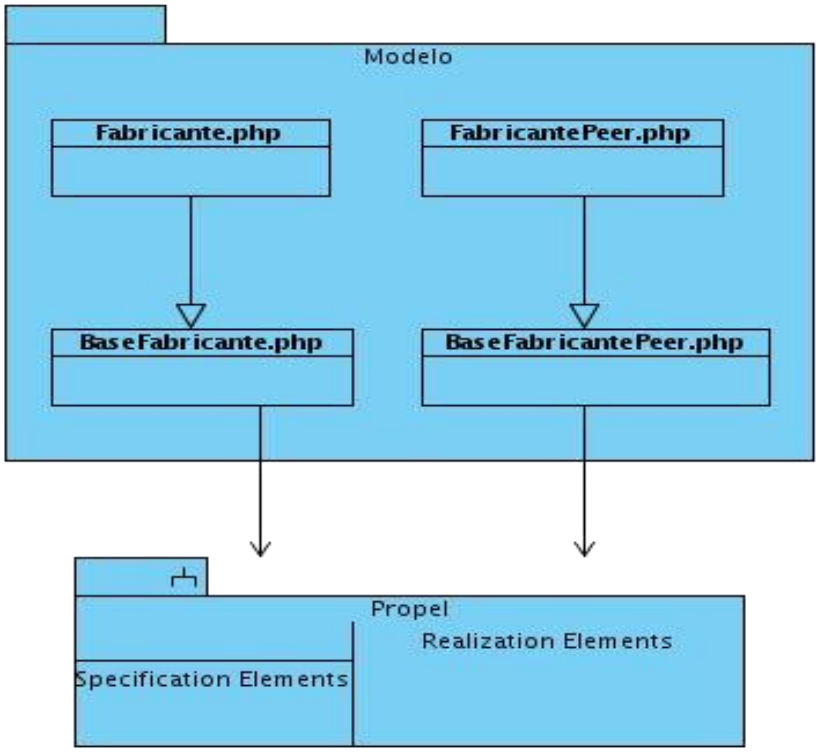


Fig. 6 Vista Lógica. Paquete Modelo

La capa de la Vista

La vista se encarga de producir páginas que se muestran como resultado de las acciones.

Paquete vista

Las páginas Web suelen contener elementos que se muestran de forma idéntica a lo largo de toda la aplicación: cabeceras de la página, el layout genérico, el pie de página y la navegación global. Normalmente sólo cambia el interior de la página. Por este motivo, la vista se separa en un layout y en una plantilla. Habitualmente, el layout es global en toda la aplicación o al menos en un grupo de páginas. Este paquete incluye las clases necesarias para presentar al usuario la lógica de la aplicación. La clase layout contiene los elementos que se muestran de forma idéntica para las páginas Web a lo largo de toda la aplicación. El layout o plantilla global como también se le conoce, almacena el código HTML común a todas las páginas de la aplicación, para no tener que repetirlo. El contenido de la plantilla se integra en el layout, o sea el layout decora la plantilla, este comportamiento es una implementación del patrón de diseño *"decorador".*

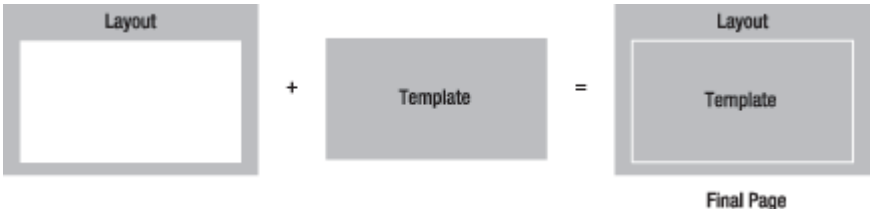


Fig. 7 Descripción del patrón de diseño decorator

Paquete Plantillas

Agrupa todas las plantillas o páginas de cada módulo que se encargan de visualizar las variables definidas en el paquete del controlador. Constituyen la presentación de los datos de la acción que se están ejecutando. Su contenido está formado por código HTML y algo de código PHP sencillo, normalmente llamadas a las variables definidas en la acción.

Subsistema ExtJS

ExtJS es una librería construida con JavaScript que proporciona una interfaz, su potencia radica en la rica colección de componentes para el diseño de GUI's del lado del cliente haciendo uso extensivo de Ajax. Entre los componentes que esta librería ofrece encontramos cuadros de diálogo, menús, tablas editables, layout, paneles, pestañas y todo lo necesario para construir atractivos desarrollos

Paquete Vista de los proyectos SACCEM y SIGICEM, contienen la clase layout con elementos que se muestran de forma idéntica para las páginas Web a lo largo de toda la aplicación y el paquete platillas el cual contiene las plantillas de cada uno de los módulos.

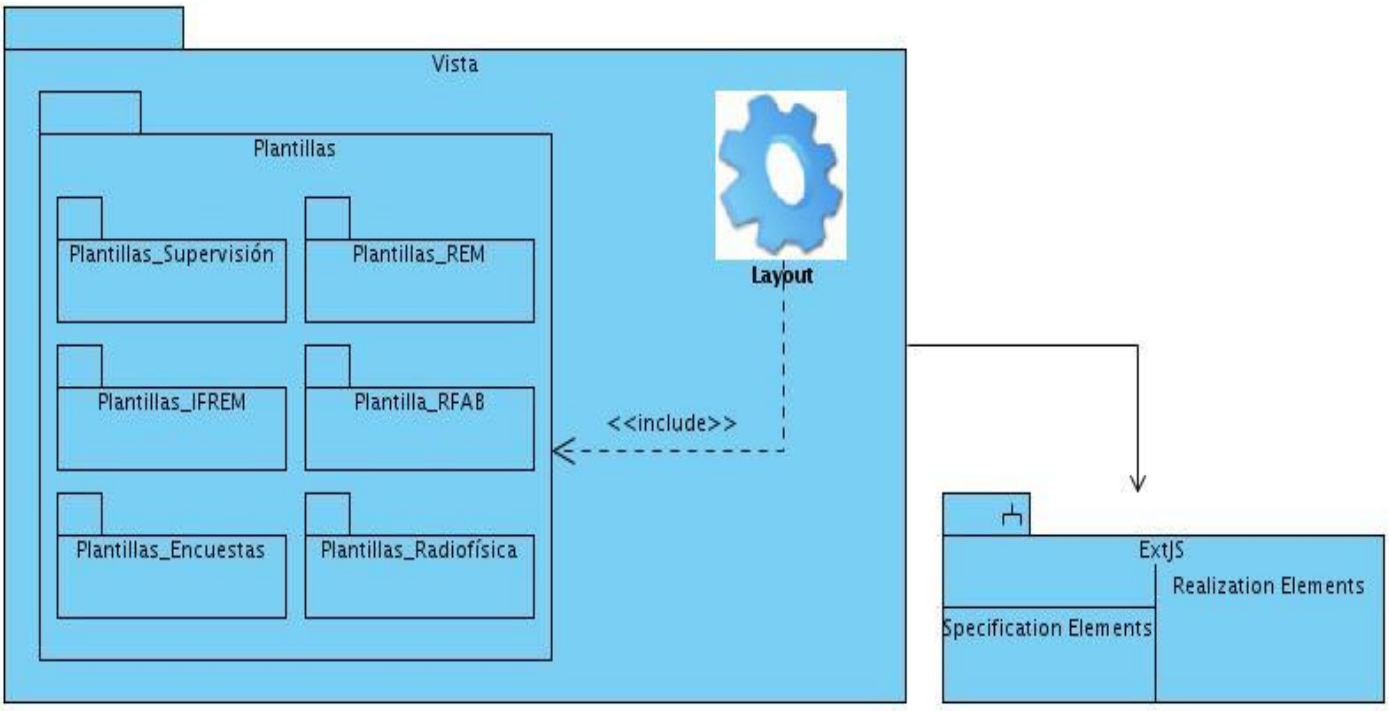


Fig. 8 Vista Lógica. Paquete Vista SACCEM

La capa del Controlador

En las aplicaciones Web el controlador se encarga de realizar numerosas tareas, suele tener mucho trabajo. Una parte importante del trabajo del controlador es común para todos los controladores de la aplicación. Entre las responsabilidades comunes se encuentran el manejo de las peticiones de la aplicación, el manejo de la seguridad, cargar la configuración de la aplicación y otras tareas similares. Por este motivo, el controlador normalmente se divide en un controlador frontal, que es único para cada aplicación, y las acciones, que incluyen el código específico del controlador de cada página.

Paquete Controlador: Agrupa las clases que implementan la lógica de la aplicación. La clase controlador frontal encapsula las tareas comunes de la lógica de cada módulo. El controlador frontal es un componente que sólo tiene código relativo al MVC, por lo que no es necesario crear uno, ya que Symfony lo genera de forma automática. Una de las principales ventajas de utilizar un controlador frontal es que ofrece un punto de entrada único para toda la aplicación. Así, en caso de que sea necesario impedir el acceso a la aplicación, solamente es necesario editar el script correspondiente al controlador frontal. Si la aplicación no dispone de controlador frontal, se debería modificar cada uno de los controladores. La primera configuración de la aplicación se encuentra en este controlador frontal, en este caso la clase index.php que contiene la definición de las constantes principales, carga el archivo general de configuración y despacha la petición.

Paquete Acciones: Encierra la clase Actions de cada módulo. Estas clases definen las acciones que incluyen el código específico del controlador de cada página.

Subsistema Componentes_Symfony

Represan todas las clases del framework Symfony que serán utilizadas durante el funcionamiento del sistema, tales como validadores de formularios, sistema de enrutamiento, componentes de seguridad y configuración, entre otros.

Paquete Controlador del proyecto SACCEM y SIGICEM, contiene la clase index.php, que constituye el controlador frontal y el paquete acciones el cual contiene las acciones de cada uno de los módulos.

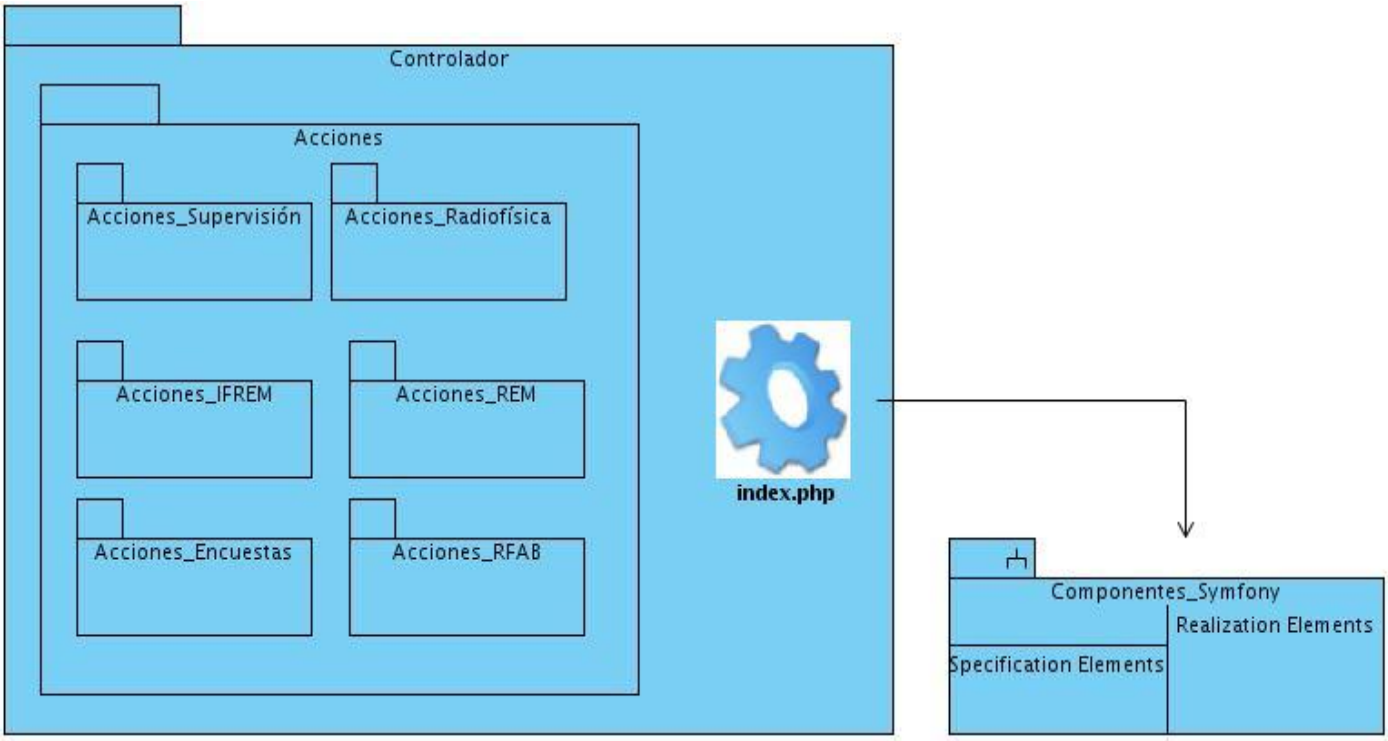


Fig. 9 Vista Lógica. Paquete Controlador SACCEM

Vista de Despliegue

La vista de despliegue describe los principales nodos físicos, ordenadores, así como los dispositivos que se necesitan para configurar la plataforma que pueda soportar la implementación del sistema. Se obtiene cuando se realiza el flujo de trabajo de Análisis y Diseño. Describe la situación de los componentes en una posible implantación del sistema de acuerdo a los requisitos iniciales. Es representada por un Diagrama de Despliegue y es un subconjunto del Modelo de Despliegue.

El Modelo Físico/de Despliegue provee un modelo detallado de la forma en la que los componentes se desplegarán a lo largo de la infraestructura del sistema. Detalla las capacidades de red, las especificaciones del servidor, los requisitos de hardware y otra información relacionada al despliegue del sistema propuesto. Esta vista representa el mapeo de componentes de software ejecutables con los nodos de procesamiento (hardware), tiene en cuenta los siguientes requerimientos: disponibilidad del sistema, rendimiento y escalabilidad.

Un nodo es un objeto físico que representa un recurso informático, este recurso generalmente dispone de datos persistentes y capacidad de proceso. Las conexiones entre nodos muestran las líneas de comunicación con las que el sistema tendrá que interactuar.

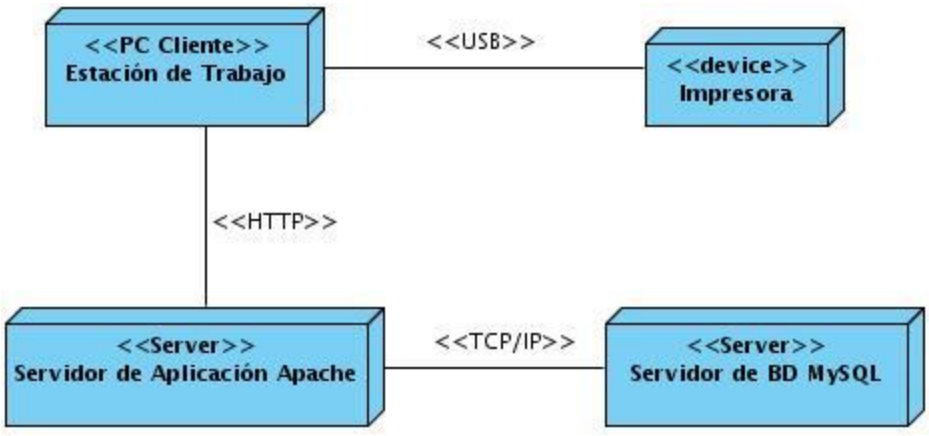


Fig.10 Vista de Despligue

Estaciones de trabajo con la Aplicación Cliente

Se refiere a las estaciones de trabajo que el usuario utilizará para acceder a la aplicación Web y transcribir sus datos.

Impresora

Este dispositivo estará a disposición de la estación de trabajo con la aplicación cliente para visualizar en una hoja impresa los datos requeridos por el cliente.

Servidor de Aplicación

Servidor de aplicación utilizado para la publicación de la aplicación; y para lograr la conexión del sistema con el la PC Cliente se utiliza HTTP (Hypertext Transfer Protocol) como protocolo de comunicación. Es la herramienta principal para ejecutar la lógica de negocio en el lado del servidor. Es el responsable de ejecutar el código de las páginas servidor. Se utiliza el servidor de aplicación Apache.

Apache está diseñado para ser un servidor Web potente y flexible que pueda funcionar en la más amplia variedad de plataformas y entornos. Apache se ha adaptado siempre a una gran variedad de entornos a través de su diseño modular. Este diseño permite a los administradores de sitios Web elegir que características van a ser incluidas en el servidor seleccionando que módulos se van a cargar, ya sea al compilar o al ejecutar el servidor. Apache puede soportar de una forma más fácil y eficiente una amplia variedad de sistemas operativos y puede personalizarse mejor para las necesidades de cada sitio Web.

Protocolos de Comunicación

Un protocolo de comunicación es un conjunto de reglas establecidas entre dos dispositivos para permitir la comunicación entre ambos.

Conexión HTTP: es el protocolo utilizado entre los browser de los clientes y el servidor Web. Este elemento de la arquitectura representa un tipo de comunicación no orientado a la conexión entre clientes y servidor.

Una alternativa de la conexión HTTP es usar “HTTP seguro” (HTTPS) que es HTTP utilizando como protocolo de transporte SSL (Secure Sockets Layer). HTTPS es un protocolo de red basado en el protocolo HTTP, destinado a la transferencia segura de datos de hipertexto. El sistema HTTPS utiliza SSL para crear un canal cifrado (cuyo nivel de cifrado depende del servidor remoto y del navegador utilizado por el cliente) más apropiado para el tráfico de información sensible que el protocolo HTTP.
Aunque quedó definido utilizar el protocolo HTTP por decisiones del cliente se realizó un estudio y análisis del protocolo HTTPS por la seguridad que ofrece el mismo.

TCP/IP

Es la base del Internet que sirve para enlazar computadoras. El protocolo **TCP/IP** es utilizado para establecer la conexión entre el servidor de aplicación y el servidor de base de datos, mientras que para poder brindar los servicios de impresión se utiliza el protocolo **USB** para la conexión entre la estación de trabajo y el dispositivo impresora.

Servidor de Base de Datos.

Se refiere al servidor que radica en cada nodo regional donde se van a estar centralizados los datos recopilados. El servidor de base de datos elegido es MySQL, que hace poco fue comprado por Sun, empresa propietaria entre otras tecnologías de JAVA y StarOffice. MySQL está disponible para MAC, Linux y Windows y los ficheros de las bases de datos son intercambiables entre los 3 sistemas operativos lo que hace que podamos tener copias de las bases de datos en distintos sistemas operativos.

Vista de Implementación

La vista de implementación contiene la organización de los módulos en términos de paquetes y capas, pueden incluirse también la trazabilidad de la vista lógica. Es representada por un diagrama de componentes o especificaciones de paquetes que son básicamente un subconjunto del modelo de despliegue. Se obtiene cuando se realiza el flujo de trabajo de Análisis y Diseño.
La vista de componentes refleja la organización de módulos de software dentro del entorno de desarrollo. Esta vista toma en cuenta los requerimientos que facilitan la programación, los niveles de reutilización y las limitaciones impuestas por el entorno de desarrollo. Para modelarla se dispone de dos elementos, los paquetes que representan una partición física del sistema y los componentes que representan la organización de los módulos de código fuente. Los paquetes de la vista lógica del modelo están mapeados con los paquetes físicos y los componentes de software, subsistemas. Esta vista se centra en la estructura de los componentes "run-time", los ejecutables del sistema y tiene en cuenta los siguientes requisitos no funcionales: rendimiento, fiabilidad, escalabilidad, integridad, seguridad, sincronización y administración del sistema. Los componentes, en un diagrama de distribución, representan los módulos físicos del código, los cuales se corresponden con los paquetes ejecutables, de esta manera, el diagrama muestra donde corre cada paquete en el sistema. Las dependencias muestran cómo los componentes se comunican con otros componentes, las cuales indican el conocimiento en la comunicación. Esta vista certifica la validez de los componentes modulares de software, los ejecutables así como la distribución de recursos informáticos.

La vista de implementación constituye una selección de los aspectos fundamentales del diagrama de componentes del sistema, el cual modela el empaquetado físico del sistema en unidades reutilizables llamadas "componentes" y sus relaciones. Un componente es una unidad física de implementación que encapsula una o más clases del diseño. Estos se pueden agrupar en subsistemas de implementación, para mayor organización y claridad del diagrama.

En resumen, la vista de implementación de la arquitectura muestra los elementos físicos reales más significativos del sistema. Symfony organiza el código fuente en una estructura de tipo proyecto y almacena los archivos del proyecto en una estructura estandarizada de tipo árbol.

Fig. 11 Estructura física real definida por el framework Symfony

Symfony establece una estructura de carpetas para sus proyectos. Dentro de un proyecto, las operaciones se agrupan de forma lógica en aplicaciones. Cada aplicación está formada por uno o más módulos. Entre las carpetas más importantes se encuentran las siguientes:

apps: Contiene un directorio por cada aplicación del proyecto.

cache: Contiene la versión cacheada de la configuración y (si está activada) la versión cacheada de las acciones y plantillas del proyecto.

config: Almacena la configuración general del proyecto

data: En este directorio se almacenan los archivos relacionados con los datos, como por ejemplo el esquema de una base de datos, el archivo que contiene las instrucciones SQL para crear las tablas e incluso un archivo de bases de datos de SQLite.

doc: Contiene la documentación del proyecto, formada por tus propios documentos y por la documentación generada por PHPdoc.

lib: Almacena las clases y librerías externas, Se suele guardar todo el código común a todas las aplicaciones del proyecto.

plugins: Almacena los plugins instalados en la aplicación

Web: Constituye la raíz Web del sistema. Los únicos archivos accesibles desde Internet son los que se encuentran en este directorio.

templates: Contiene las plantillas globales de la aplicación, es decir, las que utilizan todos los módulos. Por defecto contiene un archivo llamado layout.php, que es el layout principal con el que se muestran las plantillas de los módulos.

test: Contiene las pruebas unitarias y funcionales escritas en PHP y compatibles con el framework de pruebas de Symfony. Cuando se crea un proyecto, Symfony crea algunas pruebas básicas.

modules: Almacena los módulos que definen las características de la aplicación

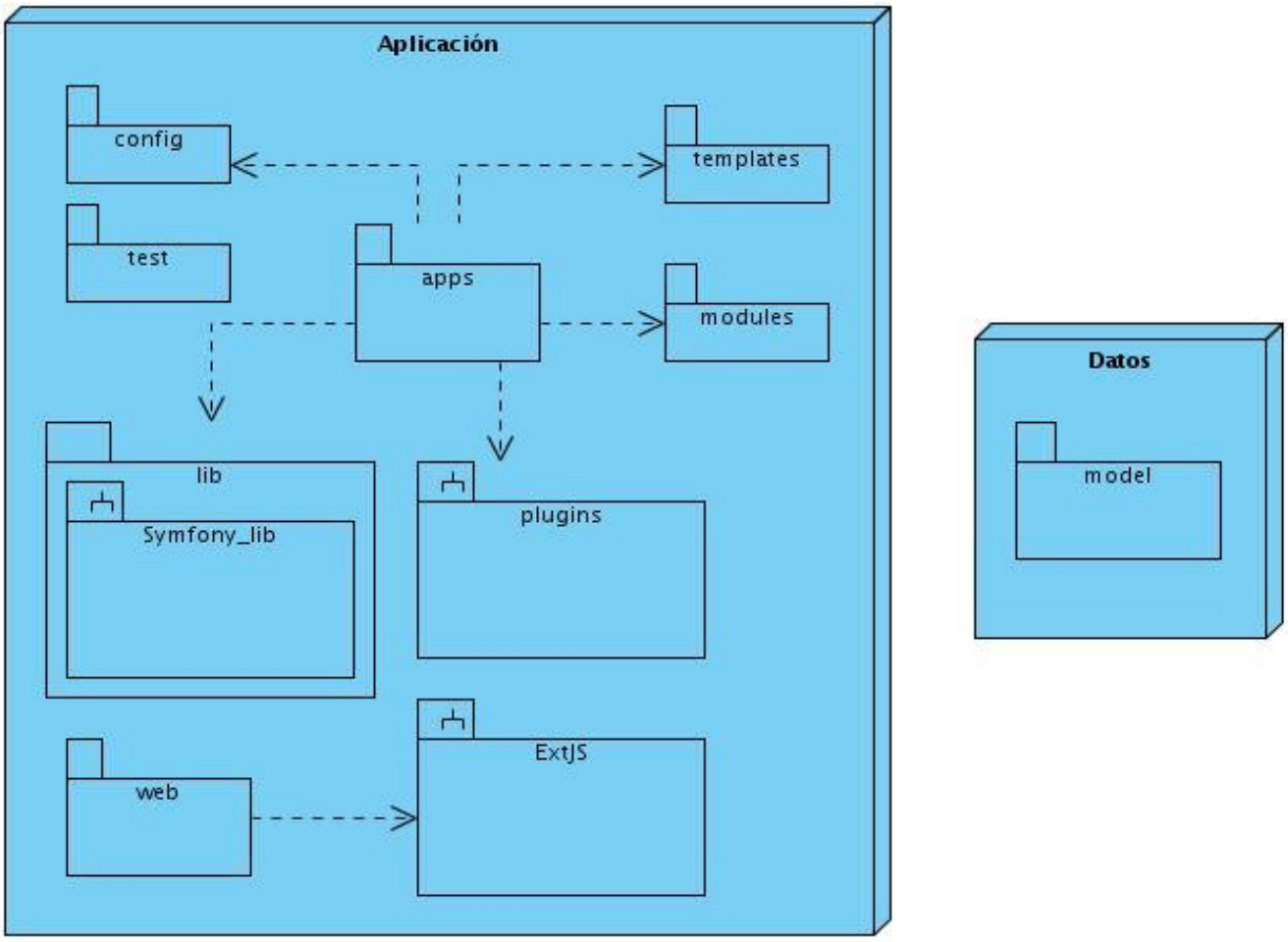


Fig. 12 Estructura General definida por el framework Symfony

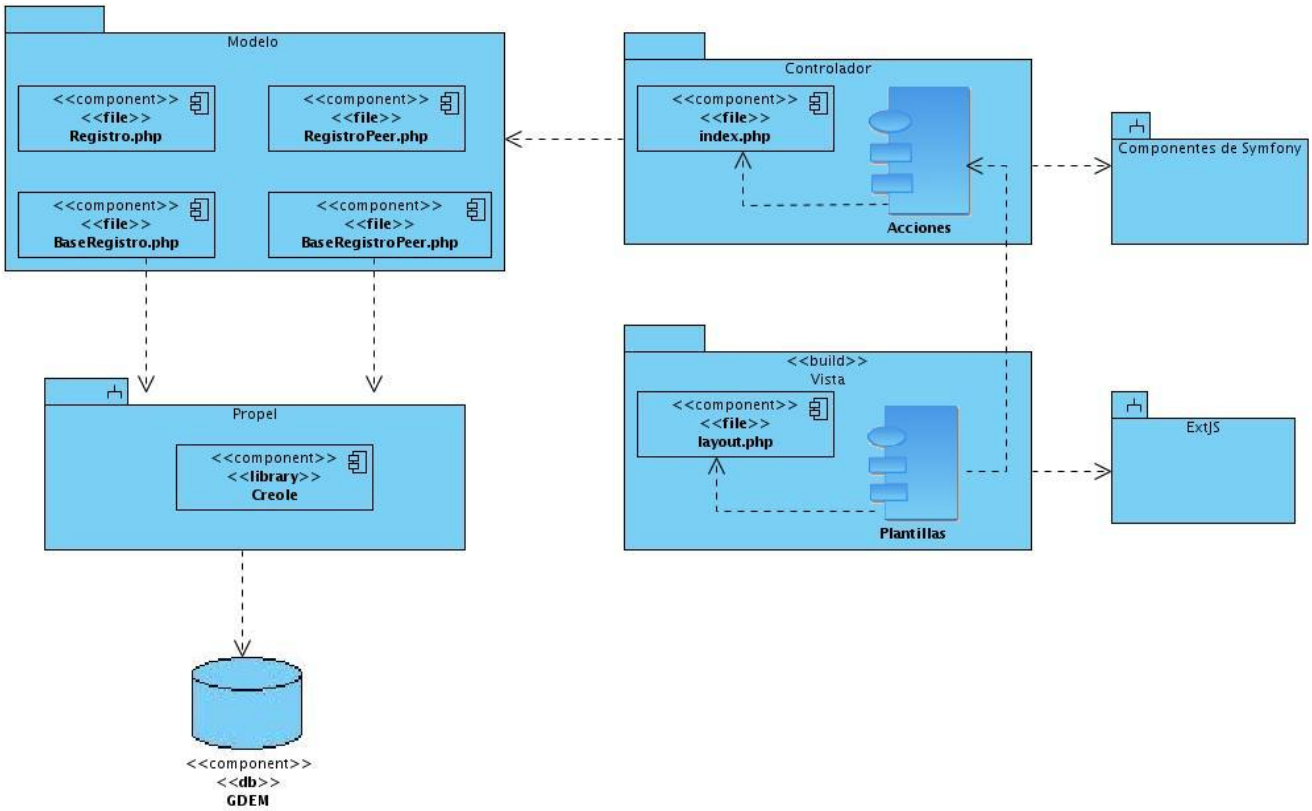


Fig. 13 Vista de Implementación detallada

Componente layout.php: Componente que implementa la clase layout del diseño. Contiene los elementos que se muestran de forma idéntica a lo largo de toda la aplicación.

Paquete Plantillas: Agrupa los componentes que implementan el código correspondiente a cada plantilla que utiliza el módulo. Se corresponde con la Carpeta Plantillas del diseño.

Paquete de componente Acciones: Encapsula los componentes actions.class.php de cada uno de los módulos, los implementan las clases Actions del diseño. Define las acciones que incluyen el código específico del controlador para cada página del módulo.

Componente security.yml: archivo de configuración que permite restringir el acceso a determinadas acciones del módulo.

Componente view.yml: archivo de configuración de las vistas de cada módulo.

Paquete Datos: Se especifica el subsistema Propel que es el ORM que utiliza Symfony el cual proporciona persistencia para los objetos y un servicio de consultas. Propel a su vez utiliza el componente Creole como sistema de abstracción de la base de datos, sistema similar a los PDO (PHP Data Object) y proporciona una interfaz entre el código PHP y el código SQL de la base de datos, permitiendo cambiar fácilmente de sistema gestor de base de datos.

Componente BD GDEM: Encapsula todos los datos del sistema.

La vista de implementación del sistema queda constituida por un grupo de componentes, y subsistemas de implementación que modelan el empaquetado físico real del sistema. Estos subsistemas de implementación representan a las carpetas del directorio del proyecto descritas anteriormente, y contienen a los ficheros modelados en término de componentes.

2.5 Conclusiones parciales del capítulo

En el capítulo se realizó un análisis acerca de la organización del sistema, así como los principales objetivos, mecanismos, pautas y restricciones arquitectónicas. Además se analizaron y definieron las principales tecnologías y herramientas informáticas para dar soporte al desarrollo del sistema. Quedó definida la plataforma LAMP a la cual se le incorporaron un conjunto de herramientas y tecnologías debido a las necesidades y metas trazadas por el GDEM.

Se realizó la descripción de la arquitectura del sistema a través de las 4+1 vistas de la arquitectura definidas por Phelipe Kruchten e incorporadas a RUP y que a su vez dicha metodología propone para realizar dicha descripción. 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 DE LA ARQUITECTURA

Introducción

El primer paso para evaluación de la arquitectura de software es conocer qué es lo que se quiere probar para poder establecer la base para la evaluación, puesto que la intención es saber qué se puede evaluar y qué no. Si las decisiones que se toman sobre la arquitectura de software determinan los atributos de calidad del sistema, entonces es posible evaluar las decisiones arquitectónicas con respecto al impacto sobre dichos atributos. La garantía de una arquitectura correcta cumple un papel fundamental en el éxito general del proceso de desarrollo de software, además del cumplimiento de los atributos de calidad del sistema. (15)

Evaluar una arquitectura de software sirve para prevenir todos los posibles problemas de un diseño que no cumple con los requerimientos de calidad y para saber que tan adecuada es la arquitectura de software definida para el sistema. De la evaluación de una arquitectura de software no se obtienen respuestas del tipo si, no, bueno o malo, sino que explica cuáles son los puntos de riesgo del diseño evaluado, es decir, fortalezas y debilidades identificadas de la arquitectura de software.

El interés se centra en determinar el momento propicio para afectar la evaluación de la arquitectura. Kazman en el año 2001 estableció dos variantes. Para la primera no es necesario que la arquitectura esté completamente especificada para efectuar la evaluación, abarcando desde las fases tempranas de diseño y a lo largo del desarrollo. Mientras mayor es el nivel de especificación, mejores son los resultados que produce la evaluación, por eso la segunda variante resulta más sólida y mejor concebida, la cual consiste en realizar la evaluación de la arquitectura cuando ésta se encuentra establecida y la implementación se ha completado, la evaluación en este punto resulta útil puesto que puede observarse el cumplimiento de los atributos de calidad asociados al sistema y cómo será su cumplimiento general.

La intención es la evaluación del potencial de la arquitectura diseñada para alcanzar los atributos de calidad requeridos. Las mediciones que se realizan sobre una arquitectura de software pueden tener distintos objetivos dependiendo de la situación en el que se encuentre el arquitecto y la aplicabilidad de las técnicas que emplea. Los objetivos que menciona Bosch en el año 2000 son tres: cualitativos, cuantitativos y máximos y mínimos teóricos.

La medición cualitativa se aplica para la comparación entre arquitecturas candidatas y tiene relación con la intención de saber la opción que se adapta mejor a cierto atributo de calidad. Este tipo de medición brinda respuestas afirmativas o negativas sin mayor nivel de detalle.

La medición cuantitativa busca la obtención de valores que permitan tomar decisiones en cuanto a los atributos de calidad de una arquitectura de software. El esquema general es la comparación con márgenes establecidos, como lo es el caso de los requerimientos de desempeño, para establecer el grado de cumplimiento de una arquitectura candidata, o tomar decisiones sobre ella. Este enfoque permite establecer comparaciones, pero se ve limitado en tanto no se conozcan los valores teóricos máximos y mínimos de las mediciones con las que se realiza la comparación.

Por último, la medición de máximo y mínimo teórico contempla los valores teóricos para efectos de la comparación de la medición con los atributos de calidad especificados. El conocimiento de los valores máximos o mínimos permite el establecimiento claro del grado de cumplimiento de los atributos de calidad.

Se han analizado y estudiado los planteamientos de Bosch (2000) y Kazman (2001) ambos indican la importancia de la especificación exhaustiva de los atributos de calidad como base para efectos de la evaluación de una arquitectura de software, el punto es entonces definir los atributos de calidad en función de sus metas y su contexto, y no como cantidades absoluta. Las principales diferencias recaen en el enfoque que utilizan para efectos de evaluación.

El método planteado por Bosch tiene como principal característica la evaluación explícita de los atributos de calidad de la arquitectura durante el proceso de diseño de la misma, sería implementar el sistema y luego establecer valores para los atributos de calidad del mismo. Este enfoque tiene la desventaja de que se destina gran cantidad de recursos y esfuerzo en el desarrollo de un sistema que no satisface los requerimientos de calidad. En este sentido plantea las técnicas de evaluación: basada en escenarios, basada en simulación, basada en modelos matemáticos y basada en experiencia.

Por su parte Kazman propone que resulta de poco interés la caracterización o medición de atributos de calidad en las fases tempranas del proceso de diseño, dado que estos parámetros son, por lo general, dependientes de la implementación. Su enfoque se orienta hacia la mitigación de riesgos, ubicando dónde un atributo de calidad de interés se ve afectado por decisiones arquitectónicas. En su estudio, presenta tres métodos de evaluación de arquitecturas de software, que son Architecture Trade-off Análisis Method (ATAM), Software Architecture Analysis Method (SAAM) y Active Intermediate Designs Review (ARID).

De los planteamientos de evaluación establecidos se tiene que la evaluación de la arquitectura de software puede ser realizada mediante el uso de diversas técnicas y métodos.

Una arquitectura de software ejerce influencia notable sobre la calidad del sistema que se implementa, de ahí la importancia de evaluarla, para determinar si cumple con los requerimientos de calidad exigidos. El poseer una buena arquitectura de software es de suma importancia, ya que ésta es el corazón de todo sistema de software y determina cuáles serán los niveles de calidad asociados al sistema.

La arquitectura es un artefacto decisivo en la calidad del software que se desarrolla. Su evaluación permite mitigar los diferentes riesgos asociados con el desarrollo del software. La evaluación permite mejorar la visión de los procesos críticos y validar las decisiones de diseño que se tomaron. Permite tomar acciones tempranas. Permite valorar los atributos de calidad sin esperar a que el software se construya.

Después de una evaluación de una arquitectura de software, se pueden tomar algunas decisiones como: si se puede seguir el proyecto con las áreas de debilidad dadas en la evaluación o si hay que reforzar la arquitectura de software o si hay que comenzar de nuevo toda la arquitectura de software.

CAPÍTULO 3

EVALUACIÓN DE LA ARQUITECTURA

3.1 Atributos de calidad

La calidad del software se define como el grado en el cual el software posee una combinación deseada de atributos. Dichos atributos constituyen requerimientos del sistema que hacen referencia a las características que el sistema debe satisfacer y es lo que se conocer como atributos de calidad. A grandes rasgos Bass establece la clasificación de los atributos de calidad en dos categorías: atributos observables vía ejecución y los no observables en vía de ejecución. [Ver Anexos 2 y 3]

Los requerimientos de calidad se ven altamente influenciados por la arquitectura del sistema. La calidad del sistema debe ser considerada a lo largo de todo el ciclo de desarrollo independientemente de que los atributos de calidad se manifiesten indistintamente durante dicho ciclo, o sea, la arquitectura determina ciertos atributos aunque existen otros que no dependen directamente de la misma. [Bass, 2004]

Los atributos de calidad se pueden organizar y descomponer de maneras diferentes, en lo que se conoce como modelos de calidad. Los modelos de calidad de software facilitan el entendimiento del proceso de la ingeniería de software [Pressman, 2002]

Entre los modelos de calidad más importantes propuestos se encuentran: McCall (1977), FURPS (1987), Dromey (1996) e ISO/IEC 9126, este último adaptado para arquitecturas de software propuesto por Losavio en el año 2003.

ISO/IEC 9126

Losavio en el año 2003 propone una adaptación del modelo ISO/IEC 9126 del año 2001 de calidad de software para efectos de la evaluación de arquitectura de software. El modelo se basa en los atributos de calidad que se relacionan directamente con la arquitectura: seguridad, funcionalidad, eficiencia, usabilidad, mantenibilidad y portabilidad. [Ver Anexos 4 y 5]

Para alcanzar un atributo de calidad, es necesario tomar decisiones de diseño arquitectónico que requieren un pequeño conocimiento de funcionalidad. Cada decisión incorporada en una arquitectura de software puede afectar potencialmente los atributos de calidad.

El modelo de calidad expuesto en la norma ISO/IEC 9126 categoriza los atributos de calidad en seis características (Funcionalidad, Fiabilidad, Usabilidad, Eficiencia, Mantenibilidad y Portabilidad), cada una de estas se dividen a su vez en subcaracterísticas que podrán ser medidas a través de métricas internas y externas.

Funcionalidad: Se refiere a la capacidad del software de proveer un conjunto de funciones que permitan cumplir con unos requerimientos o satisfacer unas necesidades implícitas. Estas funciones se deben ejecutar bajo ciertas condiciones.

Fiabilidad: Capacidad de que el software mantenga un nivel especifico de desempeño cuando es usado bajo ciertas condiciones.

Usabilidad: La capacidad del software de que sea compresible, de fácil aprendizaje, que su uso sea atractivo al usuario cuando es usado bajo ciertas condiciones.

Eficiencia: Es la capacidad del software para proporcionar una ejecución apropiada, relativo a la cantidad de recursos usados bajo ciertas condiciones.

Mantenibilidad: La capacidad que tiene el software a ser modificado. Estas modificaciones pueden ser correcciones, mejoras o adaptaciones del software a cambios en el ambiente y a requerimientos y especificaciones funcionales.

Portabilidad: La capacidad que tiene el software a ser transferido a otros ambientes. El ambiente puede ser la organización, hardware o software.

3.2 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. (15)

Técnicas de Evaluación	Instrumentos de Evaluación
Basada en Escenarios	• Profiles • Utility Tree
Basada en Simulación	• ADLs • Modelos de colas
Basada en Modelos Matemáticos	• Cadenas de Markov • Reliability y Block Diagrams
Basada en Experiencia	• Intuición y experiencia • Tradición • Proyectos similares

Tabla 4 Instrumentos asociados a las técnicas de evaluación de arquitectura de software

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, configuración, etc. 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:

- Son simples de crear y entender
- Son poco costosos y no requieren mucho entrenamiento
- Son efectivos

Actualmente las técnicas basadas en escenarios cuentan con dos instrumentos de evaluación relevantes, a saber: el Utility Tree propuesto por Kazman et al. (2001), y los Profiles, propuestos por Bosch (2000).

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. (2)

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. Algunos perfiles pueden ser usados para evaluar más de un atributo de calidad. Han sido seleccionados cinco tributos de calidad que son considerados de mayor relevancia para una perspectiva general de ingeniería de software (Bosch, 2000). Tales atributos son: desempeño (performance), mantenibilidad (maintainability), confiabilidad (reliability), seguridad externa (safety) y seguridad interna (security).

La tabla presenta para cada atributo de calidad, el perfil asociado, la forma en que se definen las categorías, el significado de los "pesos" y posibles métricas de evaluación, de acuerdo al planteamiento de Bosch (2000).

Atributo	Perfil	Categorías	Pesos
Mantenibilidad	Perfil de mantenimiento	Se organizan alrededor de las interfaces del sistema (sistema operativo, interfaces con otros sistemas). Los escenarios de cambio describen modificaciones en los requerimientos.	Indican probabilidad de ocurrencia del cambio de escenario en un período de tiempo.
Desempeño	Perfil de uso	Descompone los escenarios de uso basado en los tipos de usuario y/o interfaces del sistema.	Representan la frecuencia relativa del escenario.
Confiabilidad	Perfil de uso	Confiabilidad de los componentes, genera la confiabilidad de los escenarios de uso.	Indican la robustez del sistema.
Seguridad Interna	Perfil de seguridad	Basada en todas las interfaces del sistema	Indican la probabilidad de falla
Seguridad Externa	Perfil de peligro	Se organizan de acuerdo a documentos de certificación	Indican la probabilidad de falla u ocurrencia de consecuencias desastrosas.

Tabla 5 Perfiles, categorías y pesos asociados a atributos de calidad según Bosch (2000)

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. Se obtiene un resultado de evaluación con mayor exactitud (Bosch, 2000).

La evaluación basada en simulación utiliza una implementación de alto nivel de la arquitectura de software. La finalidad es evaluar el comportamiento de la arquitectura bajo diversas circunstancias. En el ámbito de las simulaciones se encuentra la técnica de implementación de prototipos, es utilizada para evaluar requerimientos de calidad operacional como desempeño y disponibilidad. 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 por lo que se obtiene un resultado de evaluación con mayor exactitud (Bosch, 2000).

Utiliza una implementación de alto nivel de la arquitectura de software. La finalidad es evaluar el comportamiento de la arquitectura bajo diversas circunstancias.

3.3 Métodos de evaluación

Cuando se trata de garantizar calidad en una etapa temprana, un punto de partida importante es la arquitectura del software. Ésta permite propiciar o inhibir la mayoría de los atributos de calidad esperados en él.

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. (16)

Software Architecture Analysis Method (SAAM)

Según Kazman et al. (2001), 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.

El método de evaluación SAAM se enfoca en la enumeración de un conjunto de escenarios que representan los cambios probables a los que estará sometido el sistema en el futuro. Como entrada principal, es necesaria alguna forma de descripción de la arquitectura a ser evaluada y como principales salidas de la evaluación del método SAAM son las siguientes:

- Una proyección sobre la arquitectura de los escenarios que representan los cambios posibles ante los que puede estar expuesto el sistema.
- Entendimiento de la funcionalidad del sistema, e incluso una comparación de múltiples arquitecturas con respecto al nivel de funcionalidad que cada una soporta sin modificación.

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)

Según Kazman et al. (2001), el Método de Análisis de Acuerdos de Arquitectura (Architecture Trade-off Analysis Method, ATAM) está inspirado en tres áreas distintas: los estilos arquitectónicos, el análisis de atributos de calidad y el método de evaluación SAAM, explicado anteriormente. El nombre del método ATAM surge del hecho de que revela la forma en que una arquitectura específica satisface ciertos atributos de calidad, y provee una visión de cómo los atributos de calidad interactúan con otros; esto es, los tipos de acuerdos que se establecen entre ellos.

El propósito de ATAM es evaluar las consecuencias de las decisiones arquitectónicas sobre los atributos de calidad necesarios. Es un método de identificación de riesgos, o sea detecta las áreas de riesgos potenciales en la arquitectura de un sistema. Puede hacerse en una fase temprana en el ciclo de vida del desarrollo de software. Evalúa de una forma temprana los artefactos del diseño arquitectónico. No es necesario el análisis detallado sobre los atributos de calidad, sino una identificación de tendencias; no se trata de predecir con precisión el comportamiento de un atributo de calidad, ya que es imposible en una etapa temprana del diseño tener suficiente información. El interés esta en conocer donde un atributo de interés es afectado por las decisiones del diseño arquitectónico.

Por tanto 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.

Riesgos: las decisiones de arquitectura que podría crear problemas en el futuro para algunos atributos de calidad.

No riesgos: las decisiones arquitectónicas que sean adecuadas al atributo de calidad que afectan.

Desventaja: las decisiones de arquitectura que tienen un efecto en más de un atributo de calidad.

Puntos de sensibilidad: una propiedad de uno o más componentes, y/o las relaciones entre componentes, fundamental para el logro de un determinado requisito de atributo de calidad.

El método de evaluación ATAM comprende nueve pasos, agrupados en cuatro fases.

Fase 1: Presentación	
Paso 1. Presentación del ATAM	El equipo de evaluación presenta un panorama general de los pasos de ATAM trata de establecer las expectativas, las técnicas utilizadas y de los resultados del proceso.
Paso 2. Presentación de las metas del negocio	Se realiza la descripción de las metas del negocio que motivan el esfuerzo, y aclara que se persiguen objetivos de tipo arquitectónico. Se presenta brevemente el negocio y el contexto de la arquitectura.
Paso 3. Presentación de la arquitectura	El arquitecto presenta un panorama de la arquitectura, describe la arquitectura, enfocándose en cómo ésta cumple con los objetivos del negocio.
Fase 2: Investigación y análisis	
Paso 4. Identificación de los enfoques arquitectónicos	El equipo de evaluación y el arquitecto deben detallar los planteamientos arquitectónicos descubiertos en el paso anterior. Estos elementos son detectados, pero no analizados.
Paso 5. Generación del Utility Tree	Se identifican, priorizan, definen y refinan los atributos de calidad más importantes en un formato de árbol de utilidad y que engloban la "utilidad" del sistema, especificados en forma de escenarios. Se anotan los estímulos y respuestas, así como se establece la prioridad entre ellos.
Paso 6. Análisis de los enfoques arquitectónicos	Con base en los resultados del establecimiento de prioridades del paso anterior, se analizan los elementos del paso 4. Se identifican riesgos arquitectónicos, puntos de sensibilidad y puntos de balance. Se utilizan las preguntas similares a las presentadas en el paso 5.
Fase 3: Pruebas	
Paso 7. Lluvia de ideas y establecimiento de prioridad de escenarios.	Con la colaboración de todos los involucrados, se complementa el conjunto de escenarios y se le da prioridad a los escenarios sobre la base de su importancia relativa.
Paso 8. Análisis de los enfoques arquitectónicos	Este paso repite las actividades del paso 6, haciendo uso de los resultados del paso 7. Los escenarios son considerados como casos de prueba para confirmar el análisis realizado hasta el momento.
Fase 4: Reporte	
Paso 9. Presentación de los resultados	Basado en la información recolectada a lo largo de la evaluación del ATAM, se presentan los hallazgos a los participantes. El equipo de evaluación recapitula la ATAM pasos, los resultados, y recomendaciones.

Tabla 6 Pasos del método de evaluación ATAM

Active Reviews for Intermediate Designs (ARID)

De acuerdo con Kazman et al. (2001) 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:

Fase 1: Actividades previas	
Paso 1. Identificación de los encargados de la revisión	Los encargados de la revisión son los ingenieros de software que se espera que usen el diseño, y todos los involucrados en el diseño.
Paso 2. Preparar el informe de diseño	El diseñador prepara un informe que explica el diseño. Se incluyen ejemplos del uso del mismo para la resolución de problemas reales. Esto permite al facilitador anticipar el tipo de preguntas posibles, así como identificar áreas en las que la presentación puede ser mejorada.
Paso 3. Preparar los escenarios base	El diseñador y el facilitador preparan un conjunto de escenarios base. De forma similar a los escenarios del ATAM y el SAAM.
Paso 4. Preparar los materiales	Se reproducen los materiales preparados para ser presentados en la segunda fase.
Fase 2: Revisión	
Paso 5. Presentación del ARID	Se explica los pasos del ARID a los participantes.
Paso 6. Presentación del diseño	El líder del equipo de diseño realiza una presentación, con ejemplos incluidos. El objetivo es verificar que el diseño es conveniente.
Paso 7. Lluvia de ideas y establecimiento de prioridad de escenarios	Se establece una sesión para la lluvia de ideas sobre los escenarios y el establecimiento de prioridad de escenarios. Se proponen escenarios, los cuales son sometidos a votación, y se utilizan los que resultan ganadores para hacer pruebas sobre el diseño.
Paso 8. Aplicación de los escenarios	Comenzando con el escenario que contó con más votos, el facilitador solicita pseudo-código que utiliza el diseño para proveer el servicio, y el diseñador no debe ayudar en esta tarea.
Paso 9. Resumen	Al final, el facilitador recuenta la lista de puntos tratados, pide opiniones de los participantes sobre la eficiencia del ejercicio de revisión, y agradece por su participación.

Tabla 7 Pasos del método de evaluación ARID

La selección del método basado en arquitectura se realizó a través de 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.

	ATAM	SAAM	ARID
Atributos de Calidad contemplados	• Modificabilidad. • Seguridad. • Confiabilidad. • Desempeño.	• Modificabilidad. • Funcionalidad.	• Conveniencia del diseño evaluado.
Objetivos analizados	• Estilos arquitectónicos • Documentación • Flujo de datos • Vistas arquitectónicas.	• Documentación • Vistas arquitectónicas	• Especificación de los componentes.
Etapas del proyecto en las que se aplica	• Luego de que el diseño de la arquitectura ha sido establecido.	• Luego de que la arquitectura cuenta con funcionalidad ubicada en módulos.	• A lo largo del diseño de la arquitectura
Enfoques utilizados	• Utility Tree y lluvias de ideas para articular los requerimientos de calidad. • Análisis arquitectónico que detecta puntos sensibles, puntos de balance y riesgos.	• Lluvia de ideas para escenarios y articular los requerimientos de calidad • Análisis de los escenarios para verificar funcionalidad o estimar el costo de los cambios.	• Revisiones de diseños, lluvia de ideas para obtener escenarios.

Tabla 8 Comparación entre los métodos basados en la arquitectura (Kazman 2001)

3.4 Evaluación de la arquitectura definida para el GDEM

Evaluar una arquitectura de software sirve para prevenir todos los posibles riesgos de un diseño que no cumple con los requerimientos de calidad y para saber que tan adecuada es la arquitectura de software diseñada para el sistema. Una buena regla para decidir cuando hay que realizar una evaluación de una arquitectura de software es “cuando el equipo de desarrollo comience a tomar decisiones que dependan de la arquitectura definida.

A la hora de evaluar la arquitectura de software definida para el GDEM sería válido realizarse las siguientes interrogantes:

- ¿Se ha creado una línea de base de la arquitectura?
- ¿Es adaptable y robusta?
- ¿Puede evolucionar a lo largo de la vida del producto?

La obtención correcta de la arquitectura permite la partición del sistema y que las particiones colaboren entre sí. Solidifica las interfaces entre las peticiones permitiendo la distribución del trabajo entre equipos diferentes.

Actividades de revision de arquitectura

- Revisión inicial y documentación de arquitectura física, lógica, de procesos, sus casos de uso principales y su estructura, marco de implementación.
- Una vez obtenida una representación básica del sistema, realizar un análisis del mismo con el fin de detectar oportunidades de mejora.
- Revisión de procesos de desarrollo, guías.
- Recolección de mediciones del comportamiento del sistema actual tales como: métricas de uso de recursos de los componentes del sistema: base de datos, aplicaciones, procesos.
- Análisis de los escenarios que apunten a validar requerimientos tales como escalabilidad, performance, portabilidad, usabilidad.
- Detección de posibilidades de mejora en aspectos como riesgos, arquitectura general, procesos.

Para la evaluación de la arquitectura del GDEM 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 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.

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	**Subcaracterística**	**Escenario**	**Prioridad**
Funcionalidad	Seguridad	El sistema debe restringir el acceso a los datos	A
		Un usuario no puede realizar una modificación de los datos si no posee la última versión de los mismos.	A
		Varios usuarios intentan modificar la misma información simultáneamente, el sistema solo debe dejar que modifique uno de ellos actualizando la versión de los datos.	A
		El sistema debe aceptar los datos introducidos por el usuario solo cuando estos sean los más correctos posible	A
	Adecuación	El sistema cumple con los RF y RNF solicitados por el cliente.	A
	Interoperabilidad	El sistema debe ser capaz de interactuar con otros sistemas desplegados en el SNS.	A
Fiabilidad	Tolerancia a fallos & Recuperación	Al realizarse una transacción ocurre un error en uno de los componentes, la transacción 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 & Atractividad	El sistema debe tener la capacidad de ser atractivo, entendible para el usuario.	A
Eficiencia	Usabilidad de Recursos & 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 mínimos.	A

Mantenibilidad	Cambiabilidad & Estabilidad	El sistema brida la posibilidad de cambiar de SGBD con facilidad y de emigrar hacia otro ORM dentro del mismo framework de desarrollo.	M
	Cambiabilidad & Facilidad de adaptación al cambio	El sistema debe permitir que se le puedan adicionar componentes, subsistemas, módulos o nuevas funcionalidades.	A
	Facilidad de adaptación al cambio	Los componentes pueden instalarse fácilmente en todos los ambientes donde debe funcionar, básicamente en Windows y Linux.	A
Portabilidad		Es fácil instalar las actualizaciones del sistema.	M
Leyenda: Prioridad Alta (A), Media (M), Baja (B)			

Tabla 9 Árbol de Utilidad GDEM

Evaluar las consecuencias de las decisiones arquitecturales tomadas en consideración a los atributos de calidad requeridos. 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. [Ver anexos referentes al tratamiento de los escenarios]

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.

Toma de decisiones

Se identificaron 3 riesgos descritos en los anexos del documento específicamente en las tablas de los escenarios 6,9 y 11. 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 como el SGBD.

3.5 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 6 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 14 escenarios clasificados con una prioridad entre alta, media y baja, el método arrojó finalmente 8 puntos de sensibilidad, 3 puntos de riesgos, 7 puntos de desventajas o trade off, 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

Con el objetivo de definir la arquitectura de software del GDEM se especificaron 17 requerimientos arquitectónicos del sistema, lo que propició un estudio profundo de las funcionalidades que el mismo debe cumplir.

Se planteó un soporte al desarrollo del sistema con el uso de herramientas informáticas capaces de agilizar y facilitar la construcción del mismo, quedando definida la plataforma tecnológica LAMP con la integración de otras herramientas y tecnologías a utilizar dentro de grupo y poniendo en función las buenas prácticas de la arquitectura de software contribuyendo a un entorno de desarrollo ágil seguro, confiable y a que el sistema cumpliera con las expectativas relacionadas con la estandarización en el desarrollo de las aplicaciones destinadas al SNS.

Estos requerimientos dieron paso a la definición de los principales lineamientos y mecanismos arquitectónicos a seguir dentro del grupo. Luego todos estos elementos constituyeron la base y el punto de partida para realizar la descripción de la arquitectura a través de la representación y desarrollo de las vistas de la arquitectura, las cuales proporcionaron una visión común y propuesta de solución desde las diferentes perspectivas del sistema.

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:

- La aplicación otros métodos de evaluación de la arquitectura en la etapa tardía, tales como simulación así como las pruebas de carga y estrés a la base de datos con el apoyo de herramientas como Apache JMeter.
- 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. **Bass, Len, Clement, Paul and Kazman, Rich.** *Software Architecture in Practice.* s.l. : Addison-Wesley, 2003.

2. **Clements, Paul, Kazman, Rich and Klein, Mark.** *Evaluating Software Architectures: Methods and Case Studies.* s.l. : Adison Wesley, 2000.

3. **Garlan, David.** *Software Architecture: A Roadmap.* En Anthony Finkelstein : The future engineering, ACM Press, 2000.

4. **1471-2000, IEEE Std.** *Recommended Practice for Architectural Description of Software-Intensive Systems.* 2000.

5. **González, Aleksander, et al.** *Método de Evaluación de Arquitecturas de Software Basadas en Componentes (MERCABIC).* s.l. : LISI Universidad Simón Bolívar, 2007.

6. **Microsoft.** *Estilos y Patrones en la estrategia de arquitectura.*

7. **Sarver, Toby.** *Pattern Refactoring Workshop.* 2000. 27615.

8. **EAFIT, Grupo de Investigación de Ingeniería de Software Universidad.** *La importancia de la arquitectura en el desarrollo de software de calidad.* febrero de 2005.

9. **Booch, G, Rumbaugh, J and Jacobson, I.** *El Lenguaje Unificado de Modelado.* s.l. : Adison-Wesley, 1999.

10. **Sánchez Rodríguez, Alfredo and Pompa Sourd, Frank.** *Normas para el desarrollo de aplicaciones para la salud en Cuba.* Cuba : s.n., 2007.

11. **Ronda león, Rodrigo.** *Arquitectura de Información:caminos prácticos.* 2004.

12. **Patencier, Fabien and Zaninotto, Francois.** *Symfony, la guía definitiva.* 2007.

13. **Giraldo, Luis and Zapata, Luliana.** *Herramientas de Desarrollo de Ingeniería de Software para Linux*. 2005.

14. **Sibbald, Kern and Foundation, Free Software.** *This manual documents Bacula.* Europe : s.n., Febrero 2009.

15. **Camacho, Erika, Cardeso, Fabio and Núñez, Gabriel.** *Arquitectura de Software. Guía de Estudio.* 2004.

16. **Kazman, Rick, Klein, Mark and Clements, Paul.** *ATAM: Method for Architecture Evaluation.* s.l. : Carnegie Mellon, Software Engineering Institute, 2000. CMU/SEI-2000-TR-004.

17. **Camacho, Erika, Cardeso, Fabio and Núñez, Gabriel.** *Arquitecturas de software.* 2004.

18. **Sommerville, Ian. 2005.** *Ingeniería del Software. Séptima Edición.* Madrid - España : Pearson Educación S.A, 2005. 84-7829-074-5..

19. **De Villa Cano, Raúl.** *Rol del Arquitecto de Software.* 2006.

20. **Reynoso, Carlos and Kicillof, Nicolás.** *Estilos y Patrones en la Estrategia de Arquitectura de Microsoft*. Buenos Aires, Argentina : s.n., 2004.

BIBLIOGRAFÍA

1. Booch, G., Rumbaugh, J., Jacobson, I. “El Proceso Unificado de Desarrollo de Software”. Addison-Wesley Longman, 2000.
2. Booch, G., Rumbaugh, J., Jacobson, I. “El Lenguaje Unificado de Modelado”, Addison-Wesley, 2000
3. Mendoza Sánchez, María A. “Metodologías de desarrollo de software”. Disponible en:http://www.informatizate.net/articulos/metodologias_de_desarrollo_de_software_07062004.html (23/11/2007).
4. Kruchten, P. “The Rational Unified Process: An Introduction”, Addison Wesley, 2000.
5. Sitio oficial del Visual Paradigm. Disponible en: http://www.visual-paradigm.com/product/vpuml. (21/01/2008).
6. CELIS Ismael “El ataque de los Frameworks”. Disponible en http://www.estadobeta.com/2005/11/20/el-ataque-de-los-frameworks/ (15/03/2008).
7. “Sistemas Gestores de Base de Datos”. Disponible en: http://www.error500.net/garbagecollector/bases_de_datos/sistema_gestor_de_base_de_dato.html (24/11/2007).
8. Kruchten, P. Architectural Blueprints—The “4+1” View Model of Software Architecture. 1995 [citado 2007 noviembre]; from:http://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf.
9. http://www.computer.org/portal/cms_docs_ieeecs/ieeecs/images/IBM_Rational/FINAL.SW.V12N6.42.pdf.
10. http://www.dosideas.com/noticias/42-metodologias/298-como-documentar-la-arquitectura-de-software.html.
11. WORLDWIDE INSTITUTE OF SOFTWARE ARCHITECTS. Role of the Software Architect [en línea] <http://www.wwisa.org/wwisamain/role.htm> [citado en 15 de Junio de 2008].
12. CARNEGIE MELLON UNIVERSITY What Are the Duties, Skills, and Knowledge of a Software Architect? [en línea] <http://www.sei.cmu.edu/architecture/arch_duties.html> [citado en 16 de Junio de 2008].
13. MICROSOFT CORPORATION. What Architect Job Roles Are Recognized By the
14. Microsoft Certified Architect Program? [en línea] <http://www.microsoft.com/learning/mcp/architect/specialties/default.mspx> [citado en 11 de Julio de 2007] .

BIBLIOGRAFÍA

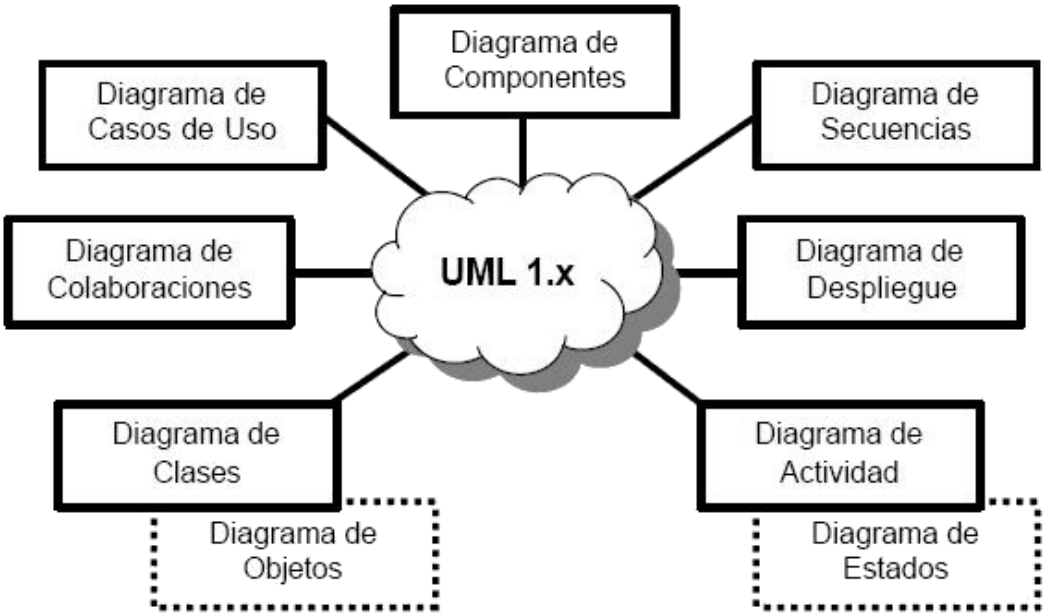
15. IBM SOFTWARE GROUP. Characteristics of a Software Architect [en línea] <http://www.ibm.com/developerworks/rational/library/mar06/eeles/> [citado el 15 marzo 2006].
16. BREDEMEYER. Architect Competency Framework [en línea] <http://www.bredemeyer.com/pdf_files/ArchitectCompetencyFramework.PDF> [citado en 16 de Junio de 2008]
17. http://www.bredemeyer.com/pdf_files/role.pdf
18. http://www.apache.org/
19. http://www.osmosislatina.com/subversion/basico.htm
20. Nord Robert, Bergey John, Blanchette Stephe, Klein Mark. Impact of Army Architecture Evaluations. CMU/SEI-2009-SR-007. Carnegie Mello University Abril 2009. <http://www.sei.cmu.edu/pub/documents/09.reports/09sr007.pdf>.
21. Documentación del SEI en la universidad Carnegie Mellon <http://www.sei.cmu.edu/publications/publications.html>.
22. Carriere, J., Kazman, R., Woods, S. (2000). Toward a Discipline of Scenariobased Architectural Engineering. Software Engineering Institute, Carnegie Mellon University. Disponible en: http://www.sei.cmu.edu/staff/rkazman/annals-scenario.pdf
23. Peralta Arturo. Calidad en sistemas basados en componentes. escuela Superior de Informática Universidad de Castilla, La Mancha. Mayo 2008.
24. Kazman, R., Clements, P., Klein, M.: Evaluating Software Architectures: Methods and Case Studies. Addison-Wesley 348 pages. ISBN 0-201-70482-X. USA. (2002).
25. http://www.sei.cmu.edu/architecture/ata_method.html
26. http://www.sei.cmu.edu/pub/documents/96.reports/pdf/tr036.96.pdf
27. Pressman, Roger, Ingeniería del Software. Un enfoque práctico. Editorial Félix Varela, La Habana, 2005
28. RUP, 2003 Rational Unified Process, IBM, 2003
29. Bass, Len, Clement, Paul and Kazman, Rich. Software Architecture in Practice. s.l.: Addison-Wesley, 2003.
30. Clements, Paul, Kazman, Rich and Klein, Mark. Evaluating Software Architectures: Methods and Case Studies. s.l.: Adison Wesley, 2000.
31. Garlan, David. Software Architecture: A Roadmap. En Anthony Finkelstein : The future engineering, ACM Press, 2000.

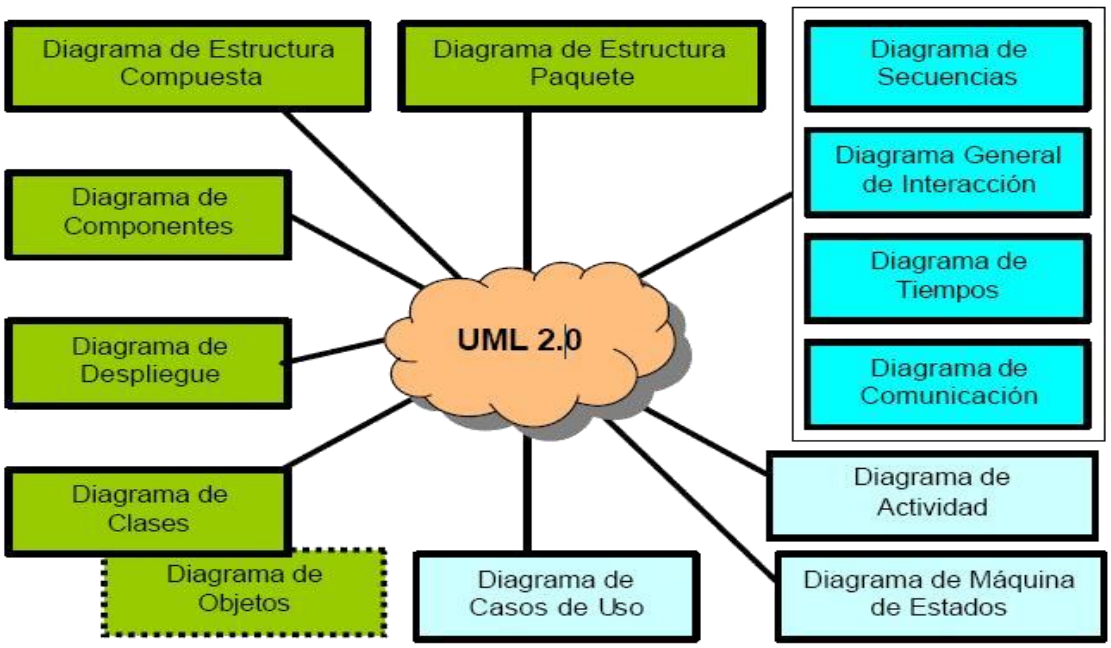
BIBLIOGRAFÍA

32. 1471-2000, IEEE Std. Recommended Practice for Architectural Description of Software-Intensive Systems. 2000.
33. González, Aleksander, et al. Método de Evaluación de Arquitecturas de Software Basadas en Componentes (MERCABIC). s.l. : LISI Universidad Simón Bolívar, 2007.
34. Microsoft. Estilos y Patrones en la estrategia de arquitectura, 2005
35. Sarver, Toby. Pattern Refactoring Workshop. 2000. 27615.
36. EAFIT, Grupo de Investigación de Ingeniería de Software Universidad. La importancia de la arquitectura en el desarrollo de software de calidad. febrero de 2005.
37. Booch, G, Rumbaugh, J and Jacobson, I. El Lenguaje Unificado de Modelado. s.l. : Adison-Wesley, 1999.
38. Sánchez Rodríguez, Alfredo and Pompa Sourd, Frank. Normas para el desarrollo de aplicaciones para la salud en Cuba. Cuba: s.n., 2007.
39. Ronda león, Rodrigo. Arquitectura de Información:caminos prácticos. 2004.
40. Patencier, Fabien and Zaninotto, Francois. Symfony, la guía definitiva. 2007.
41. 13. Giraldo, Luis and Zapata, Luliana. Herramientas de Desarrollo de Ingeniería de Software para Linux. 2005.
42. Sibbald, Kern and Foundation, Free Software. This manual documents Bacula. Europe : s.n., Febrero 2009.
43. Camacho, Erika, Cardeso, Fabio and Núñez, Gabriel. Arquitectura de Software. Guía de Estudio. 2004.
44. Kazman, Rick, Klein, Mark and Clements, Paul. ATAM: Method for Architecture Evaluation. s.l. : Carnegie Mellon, Software Engineering Institute, 2000. CMU/SEI-2000-TR-004.
45. Camacho, Erika, Cardeso, Fabio and Núñez, Gabriel. Arquitecturas de software. 2004.
46. Sommerville, Ian. 2005. Ingeniería del Software. Séptima Edición. Madrid - España : Pearson Educación S.A, 2005. 84-7829-074-5..
47. De Villa Cano, Raúl. Rol del Arquitecto de Software. 2006.
48. Reynoso, Carlos and Kicillof, Nicolás. Estilos y Patrones en la Estrategia de Arquitectura de Microsoft. Buenos Aires, Argentina : s.n., 2004.

ANEXOS

Anexo # 1 Diagramas de UML 1.0 y UML 2.0





Anexo # 2 Descripción de los atributos de calidad observables vía ejecución.

Atributo de Calidad	Descripción
Disponibilidad (Availability)	Es la medida de disponibilidad del sistema para el uso. (Barbacci, 1995)
Confidencialidad (Confidentiality)	Es la ausencia de acceso no autorizado a la información. (Barbacci, 1995)
Funcionalidad (Functionality)	Habilidad del sistema para realizar el trabajo para el cual fue concebido. (Kazman, 2001)
Desempeño (Performance)	Es el grado en el cual un sistema o componente cumple con sus funciones designadas, dentro de ciertas restricciones dadas, como velocidad, exactitud o uso de memoria. (IEEE 610.12) El desempeño de un sistema se refiere a aspectos temporales del comportamiento del mismo. Se refiere a la capacidad de respuesta, ya sea el tiempo requerido para responder a aspectos específicos o el número de eventos procesados en un intervalo de tiempo. (Smith, 1993). Se refiere además a la cantidad de comunicación e interacción existente entre los componentes del sistema (Bass, 1998).
Confiabilidad (Reliability)	Es la medida de la habilidad de un sistema a mantenerse operativo a lo largo del tiempo. (Barbacci, 1995)
Seguridad Externa (Safety)	Ausencia de consecuencias catastróficas en el ambiente. Es la medida de ausencia de errores que generan pérdidas de información (Barbacci, 1995)
Seguridad Interna (Security)	Es la medida de la habilidad del sistema para resistir a intentos de uso no autorizados y negación de servicio, mientras sirve a los usuarios legítimos (Kazman, 2001).

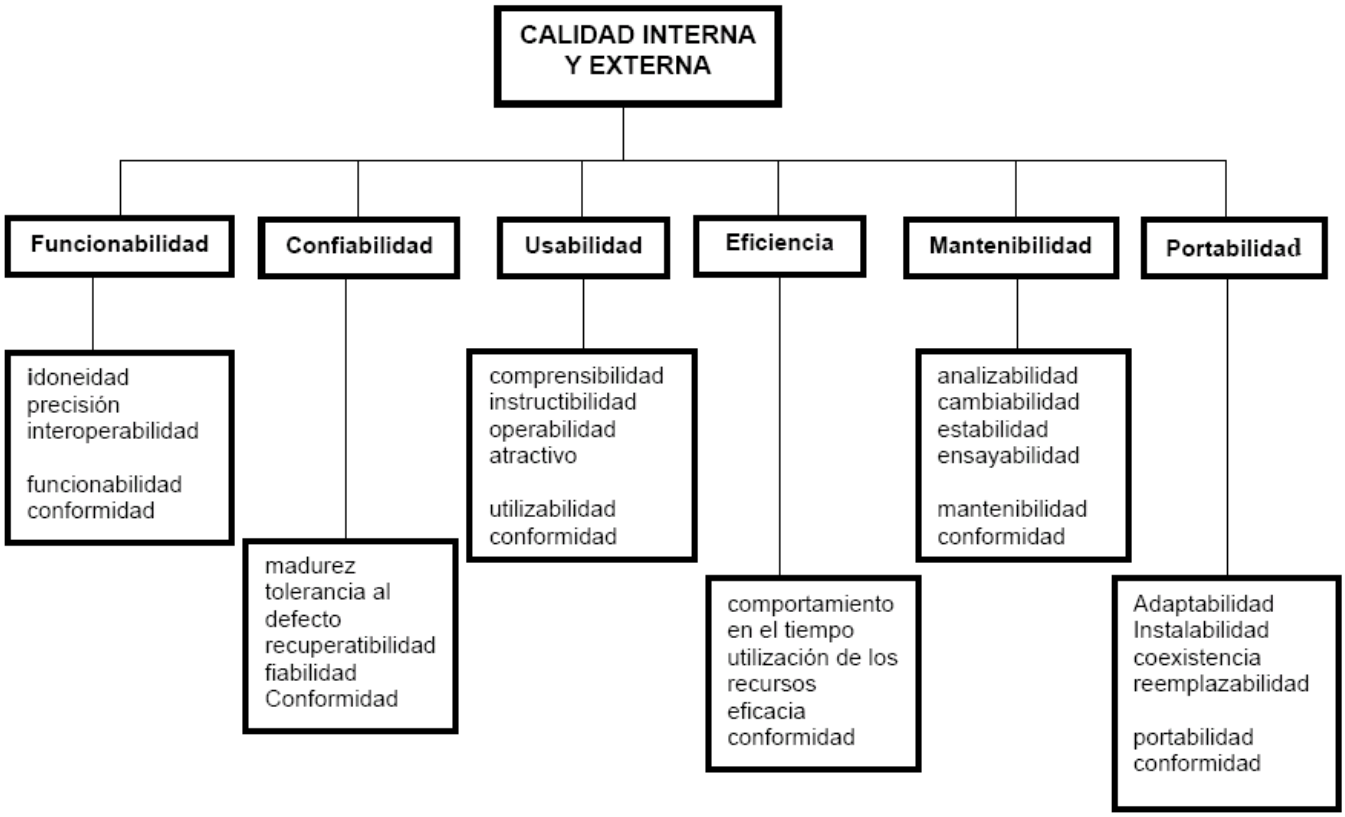
Anexo # 3 Tabla 3 Descripción de los atributos de calidad no observables vía ejecución.

Atributo de Calidad	Descripción
Configurabilidad (Configurability)	Posibilidad que se otorga a un usuario experto a realizar ciertos cambios al sistema (Bosch, 1999).
Integrabilidad (Integrability)	Es la medida en que trabajan correctamente componentes del sistema que fueron desarrollados separadamente al ser integrados. (Bass, 1998).
Integridad (Integrity)	Es la ausencia de alteraciones inapropiadas de la información (Barbacci, 1995).
Interoperabilidad (Interoperability)	Es la medida de la habilidad de que un grupo de partes del sistema trabajen con otro sistema. Es un tipo especial de integrabilidad. (Bass, 1998).
Modificabilidad (Modifiability)	Es la habilidad de realizar cambios futuros al sistema (Bosch, 1999).
Mantenibilidad (Maintainability)	Es la capacidad de someter a un sistema a reparaciones y evolución (Barbacci, 1995). Capacidad de modificar el sistema de manera rápida y a bajo costo (Bosch, 1999).
Portabilidad (Portability)	Es la habilidad del sistema para ser ejecutado en diferentes ambientes de computación. Estos ambientes pueden ser hardware, software o una combinación de los dos. (Kazman, 2001).
Reusabilidad (Reusability)	Es la capacidad de diseñar un sistema de forma tal que su estructura o parte de sus componentes puedan ser reutilizados en futuras aplicaciones. (Bass, 1998)
Escalabilidad (Scalability)	Es el grado con el que se pueden ampliar el diseño arquitectónico, de datos o procedimental (Pressman, 2002). Es la capacidad de un sistema de admitir una mayor carga de trabajo mediante la adición incremental de recursos sin tener que cambiar básicamente el diseño ni la arquitectura del mismo.
Capacidad de Prueba (Testability)	Es la medida de la facilidad con la que el software, al ser sometido a una serie de pruebas, puede demostrar sus fallas. Es la probabilidad de que, asumiendo que tiene al menos una falla, el software fallará en su próxima ejecución de prueba (Bass, 1998).

Anexo # 4 Principales atributos de calidad definidos por estándares

IEEE Std 1061	ISO Std 9126	MITRE Guide to Total Software Quality Control	
Eficiencia	Funcionalidad	Eficiencia	Integridad
Funcionalidad	Confiabilidad	Confiabilidad	Supervivenciabilidad
Mantenibilidad	Usabilidad	Usabilidad	Corretitud
Portabilidad	Eficiencia	Mantenibilidad	Verificabilidad
Confiabilidad	Mantenibilidad	Expandibilidad	Flexibilidad
Usabilidad	Portabilidad	Interoperabilidad	Portabilidad
		Reusabilidad	

Anexo #5 Atributos de calidad de la norma ISO/IEC 9126



Tratamiento de los escenarios

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 de ambos sistemas, se han definido varias estrategias. Se restringe según el rol y los servicios instalados al usuario, el acceso a la base de datos, y en esta se guarda la información que se refiere al tipo de servicio específico, permitiendo así no tener redundancia.			

Escenario #: 2	Un usuario no puede realizar una modificación de los datos si no posee la última versión de los mismos.			
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	La integridad de la Información resulta extremadamente importante en los sistemas informáticos. Para mantener la integridad de la información en ambos sistemas, se han definido varias estrategias. Se restringe a los usuarios a que si no cuenta con la última versión de los datos no puede modificarlos.			

Escenario #: 3	Varios usuarios intentan modificar la misma información simultáneamente, el sistema solo debe dejar que modifique uno de ellos actualizando la versión de 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
		S3		
Explicación				

Escenario #:4	El sistema debe aceptar los datos introducidos por el usuario solo cuando estos sean los más correctos posible.			
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
		S4		
Explicación	Se lleva a cabo la validación de los formularios para alcanzar elevados niveles de confiabilidad sobre los datos con se opera. 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 antes de enviar el formulario. Además quedó definido enviar los datos por POST, limpiar las url, para alcanzar una mayor integridad de los mismos.			

Escenario #: 5	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
		S5		
Explicación	La arquitectura propuesta satisface con los RF y RNF plateados por los clientes, brinda soporte a las acciones posibilitando la gestión de la información de los equipos médicos y la gestión de la tecnología médica, centrándose en las necesidades del usuario.			

Escenario #:6	El sistema debe ser capaz de interactuar con otros sistemas desplegados en el SNS.			
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 SNS.			
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 SNS como es el caso del RIS y a la hora de tomar las decisiones arquitectónicas se pensó en dicho detalle.			

Escenario #: 7	Al realizarse una transacción ocurre un error en uno de los componentes, la transacción 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 tras la ejecución de una transacción.			
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.			

Escenario #:8	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 arquitectónicas	Riesgo	Punto de Sensibilidad	Punto de Intercambio	No Riesgo
			T3	
Explicación	La combinación definida de framework a utilizar permitirá obtener aplicaciones entendibles, atractivas para el usuario, con interfaces amigables.			

Escenario #: 9	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	El usuario necesita realizar actualizaciones desde OBE al centro provincial después de haber estado sin conexión un período determinado de tiempo			
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.			

Escenario #:10	Los tiempos de respuesta a las peticiones realizadas por los usuarios deben ser mínimos.			
Atributo(s)	Eficiencia, Rendimiento-Tiempo de Respuesta.			
Ambiente	Operación normal.			
Estímulo	El usuario necesita realizar una determinada acción en el sistema.			
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 SGBD definido es ligero, se implementan estrategias de réplica para balancear la carga del sistema. El mecanismo de cache de configuración que facilita Symfony posibilita que el tiempo de respuesta a las peticiones sea relativamente rápido, garantiza que las páginas Web respondan rápido.			

<table>
<tr><td>Escenario # 11</td><td colspan="4">El sistema brida la posibilidad de cambiar de SGBD con facilidad. Debe existir la posibilidad de emigrar hacia otro ORM dentro del mismo framework de desarrollo.</td></tr>
<tr><td>Atributo(s)</td><td colspan="4">Mantenibilidad-Cambiabilidad-Estabilidad, Rendimiento-Tiempo de Respuesta</td></tr>
<tr><td>Ambiente</td><td colspan="4">Operación normal.</td></tr>
<tr><td>Estímulo</td><td colspan="4">Necesidad del cliente o del equipo de desarrollo emigrar hacia otro SGBD.</td></tr>
<tr><td>Respuesta</td><td colspan="4">No hay cambios significativos en el sistema ni en las decisiones arquitectónicas ya establecidas.</td></tr>
<tr><td>Decisiones arquitectónicas</td><td>Riesgo</td><td>Punto de Sensibilidad</td><td>Punto de Intercambio</td><td>No Riesgo</td></tr>
<tr><td></td><td>R3</td><td></td><td>T6</td><td></td></tr>
<tr><td>Explicación</td><td colspan="4">En el desarrollo de software la ocurrencia de cambios es algo normal, por una necesidad de los desarrolladores e incluso de los clientes.
Al desarrollar la plataforma usando como framework symfony, no existirán graves problemas. Este framework es una clara implementación del acceso y abstracción de los datos en el modelo, con el uso de Propel como ORM y a su vez Propel usa Creole como una capa de abstracción, este último permite realizar un cambio de SGBD en cualquier momento, no se debe reescribir ni una línea de código, ya que tan sólo es necesario modificar un parámetro en un archivo de configuración.</td></tr>
</table>

Escenario #: 12	El sistema debe permitir que se le puedan adicionar componentes, subsistemas, 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
		S6		
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. En este sentido, el usuario es capaz de incorporar o quitar funcionalidades al sistema según la acción que desee hacer.			

Escenario #: 13	Los componentes pueden instalarse fácilmente en todos los ambientes donde debe funcionar, básicamente en Windows y Linux.			
Atributo(s)	Mantenibilidad-Facilidad de adaptación al cambio, Portabilidad			
Ambiente	Operación normal.			
Estímulo	Despliegue de las aplicaciones.			
Respuesta	La arquitectura definida soporta que ambos sistemas puedan desplegarse en ambos sistemas operativos. Se escogió como lenguaje de programación PHP y como SGBD MySQL.			
Decisiones arquitectónicas	Riesgo	Punto de Sensibilidad	Punto de Intercambio	No Riesgo
		S7		
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.			

<table>
<tr><td>Escenario #: 14</td><td colspan="4">Es fácil instalar las actualizaciones del sistema</td></tr>
<tr><td>Atributo(s)</td><td colspan="4">Portabilidad</td></tr>
<tr><td>Ambiente</td><td colspan="4">Operación normal.</td></tr>
<tr><td>Estímulo</td><td colspan="4">Necesidad de realizar actualizaciones al sistema.</td></tr>
<tr><td>Respuesta</td><td colspan="4">El sistema una vez desplegado permite realizar las actualizaciones pertinentes.</td></tr>
<tr><td>Decisiones arquitectónicas</td><td>Riesgo</td><td>Punto de Sensibilidad</td><td>Punto de Intercambio</td><td>No Riesgo</td></tr>
<tr><td></td><td></td><td>S8</td><td></td><td></td></tr>
<tr><td>Explicación</td><td colspan="4">El sistema una vez instalado y puesto en ejecución brinda la posibilidad de realizar las transferencias de archivos de actualización de forma incremental mediante el empleo de la herramienta Rsync que brinda Symfony mediante SSH. Rsync es una utilidad de la línea de comandos que permite realizar una transferencia incremental de archivos de forma muy rápida, además de que es una herramienta de software libre.
Symfony utiliza SSH conjuntamente con rsync para hacer más segura la transferencia de datos. La mayoría de servicios de hosting soportan el uso de SSH para aportar más seguridad a la transferencia de archivos hasta sus servidores.</td></tr>
</table>

GLOSARIO

Infomed: Red Telemática de Salud en Cuba.

MIC: Ministerio de la Informática y las Comunicaciones en Cuba.

MINSAP: Ministerio de Salud Pública en Cuba.

OBG: Organización Básica de Electromedicina

Stakeholders: Personas u organizaciones que están activamente implicadas en el negocio, ya sea porque participan en él o porque sus intereses se ven afectados con los resultados del proyecto.

ADL: Lenguaje de descripción de arquitectura cuyas siglas se derivan de su nombre en inglés Architecture Descripction Language y su función como su nombre lo indica es describir una arquitectura de software.

SVN: Sistema de control de versiones cuyas siglas vienen dadas por su nombre en inglés Concurrent Version System.

Framework: Se conoce como marco de trabajo y constituye un conjunto de conceptos, metodologías y herramientas de administración y diseño para el desarrollo de forma estandarizada de una aplicación.

Herramientas CASE: Conjunto de aplicaciones informáticas orientadas al incremento de la productividad en el desarrollo de software, las siglas CASE vienen dadas por su nombre en inglés Computer Arded Software Engineering que se conoce como Ingeniería de Software Asistida por Computadoras.

IEEE: Asociación técnico profesional mundial dedicada entre otros aspectos a la estandarización, su siglas en vienen dadas por su nombre en inglés The Institute of Electrical and Electronics Engineers

IBM: Empresa internacional mundialmente conocida por la venta de tecnología informática cuyas siglas en vienen dadas por su nombre en inglés Internacional Business Machina Corporation.

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

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

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

GUI: Interfaz gráfica para el usuario.

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

Herramienta ORM (Mapeo Objeto-Relacional): ayuda a reducir la llamada diferencia de impedancia Objeto-Relacional.