Universidad de las Ciencias Informáticas

Facultad 6

Título: Sistema para la Gestión de Información de Profesores y Estudiantes de la Facultad 6: Módulo Arquitectura.

Trabajo de Diploma para optar por el título de
Ingeniero Informático

Autores: Esley León Valdés
Duanis Sotolongo Vázquez

Tutores: Ing. Yamila Mateu Romero
Ing. Reynaldo Rosado Roselló

Co-tutores: Lic. Haydee Noemí Vidal Carrillo
Lic. Sandy Henríquez Villafruela

Ciudad de La Habana, Julio, 2009

Declaración de autoría

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

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

Esley León Valdés

_____

Firma del Autor

Duanis Sotolongo Vázquez

_____

Firma del Autor

Yamila Mateu Romero

_____

Firma del Tutor

Reynaldo Rosado Roselló

_____

Firma del Tutor

Contactos

Tutores:
Ing. Yamila Mateu Romero.
Universidad de las Ciencias Informáticas, Habana, Cuba.
e-mail: ymateu@uci.cu

Ing. Reynaldo Rosado Roselló.
Universidad de las Ciencias Informáticas, Habana, Cuba.
e-mail: rrosado@uci.cu

Co-tutores:
Lic. Haydee Noemí Vidal Carrillo
Universidad de las Ciencias Informáticas, Habana, Cuba.
e-mail: noemi@uci.cu

Lic. Sandy Henríquez Villafruela
Universidad de las Ciencias Informáticas, Habana, Cuba.
sandyh@uci.cu

Resumen

Con el surgimiento de la era digital, el constante avance de las tecnologías y el almacenamiento digital, surgen grandes posibilidades de implementación de sistemas informáticos capaces de manejar grandes volúmenes de datos, cuyo uso se hace necesario debido a los crecientes volúmenes de información que se manejan hoy día en todo el mundo, incluida Cuba. La UCI, compuesta por diez facultades, es una de las universidades cubanas que también maneja grandes volúmenes de información. Específicamente la facultad 6 se ha planteado implementar un sistema informatizado compuesto por seis módulos para digitalizar la información de los estudiantes y profesores. El presente documento presenta la propuesta de arquitectura para la integración de los módulos de dicho sistema. Identificando los patrones y estilos arquitectónicos más convenientes a utilizar, así como la definición de las herramientas y tecnologías a emplear según sus características y ajuste al objetivo general que se desea cumplir. Se hará referencia a los conceptos vinculados al objeto de estudio, los tipos de arquitectura y los patrones arquitectónicos utilizados. Además de realizar una descripción de la arquitectura, también se evaluará la misma una vez realizado el estudio de distintos métodos para evaluar un diseño arquitectónico y la selección del más adecuado. También se evaluará sus costos y beneficios y se hará un análisis de la solución propuesta. Se integrarán los módulos del sistema para garantizar un mejor flujo de la información y se diseñará la interfaz principal del sistema.

Palabras claves:

SIGIPE, Arquitectura, integración, gestión de información.

Tabla de Contenidos

Resumen ..... **I**

Introducción **1**

CAPÍTULO 1: Fundamentación teórica **4**

Introducción **4**

1.1 Arquitectura de Software **5**

1.2.1 Estilos arquitectónicos **7**

1.2.2 Patrones de Software **8**

Patrones arquitectónicos 8

Patrones de diseño 10

1.3 Lenguajes de Descripción Arquitectónica **14**

1.4 Lenguaje Unificado de Modelado (UML) **16**

1.5 Metodología de desarrollo de software **18**

1.6 Lenguajes, tecnologías, herramientas de soporte al desarrollo y herramientas CASE **22**

1.6.1 Plataforma y Framework. **25**

1.6.2 Gestor de Base de datos **27**

1.7 El arquitecto de software y los artefactos **28**

1.8 Conclusiones **29**

CAPÍTULO 2: Descripción de la arquitectura **30**

Introducción **30**

2.1 Análisis de los Requerimientos **31**

2.2 Vista de Casos de Uso **32**

2.3 Vista lógica **40**

Capa de Presentación o Vista: 41

Capa Controlador: 42

Capa Modelo: 43

2.4 Vista de Implementación 44

2.5 Vista de Despliegue 52

2.6 Modelo de Datos 54

2.7 Metas y restricciones arquitectónicas 57

2.8 Estándar de codificación a utilizar en la implementación del sistema. 58

2.9 Definición de los estilos CSS a utilizar en el diseño de la aplicación del sistema. 60

2.10 Conclusiones 62

CAPÍTULO 3: Evaluación de la Arquitectura. 63

Introducción 63

3.1 Evaluación de la arquitectura de software 64

3.2 Técnicas de evaluación 65

3.3 Método de evaluación de la arquitectura. 66

3.4 Árbol de utilidades para la evaluación de la arquitectura propuesta. 70

3.5 Conclusiones 76

Conclusiones generales 77

Referencias bibliográficas 79

Bibliografía 80

Anexos 82

Glosario de términos 85

Introducción

El avance en las tecnologías de la información es sin duda una de los grandes pasos que está determinando el desarrollo técnico de la humanidad en varias esferas. Con el surgimiento de la era digital, el constante avance de las tecnologías y el almacenamiento de la información en soporte digital, surgen grandes posibilidades de implementación de sistemas informáticos que respondan a las demandas de los clientes y en las que se aprovechen además, las facilidades de almacenamiento, procesamiento y distribución de la información electrónica que ofrece la revolución tecnológica y la creciente expansión de las redes. El natural crecimiento de los volúmenes de información en el que se ve envuelta la sociedad mundial trae consigo una dificultad para el almacenamiento, procesamiento, disponibilidad y distribución de la misma, haciendo que la humanidad sea cada vez más dependiente de sistemas informáticos especializados en la gestión de información como alternativa al almacenamiento en papel. Últimamente la demanda de aplicaciones robustas tanto para escritorios como compartidas en la red que garanticen la integridad, seguridad y disponibilidad de la información de los usuarios ha crecido considerablemente. Esto hace necesario además, la participación de técnicos, profesionales experimentados, plataformas de hardware y software, el uso de herramientas; así como de técnicas que propicien el desarrollo de aplicaciones consistentes que cubran las necesidades requeridas por los usuarios.

Cuba no se encuentra exenta de las demandas de desarrollo de software, tanto nacional como internacionalmente y ha hecho de ésta una tarea de gran prioridad debido a la alta perspectiva económica que posee. Instituciones como el In*stituto Superior Politécnico José Antonio Echeverría (CUJAE)*, la *Universidad de la Habana (UH)* y la *Universidad de Ciencias Informáticas (UCI)* han contribuido no solo a la formación de profesionales en el campo de la informática sino también a la producción de software dedicado a diferentes esferas como la medicina, la salud, el deporte, la educación, etc. La UCI, universidad surgida como un centro para la formación continua e integral de profesionales en la informática, constituye uno de los pilares fundamentales para el desarrollo de aplicaciones de gestión de información en Cuba. La misma se centra en el desarrollo de proyectos en más de 30 polos productivos destacando resultados en las esferas de salud, educación, software libre, tele-formación, sistemas legales, realidad virtual y automatización. La facultad 6, una de las 10 con las que cuenta la gran cuidad digital, también necesita del desarrollo de aplicaciones para el control y la gestión de información de sus estudiantes. Este control y gestión de información se lleva a cabo a través de aplicaciones como el *Sistema de Gestión Académica (Akademos)*, el Sis*tema para la Reservación de Pase*, el *Sistema para la Reservación de Transporte*, etc.

Además del control de la información de los estudiantes, es necesario el control de la información de los profesores, proyectos productivos, trabajos de diploma, residencia y actividades de extensión universitaria que se desarrollan en la facultad como parte del proceso de formación docente. La facultad actualmente se encuentra desarrollando el *Sistema para la Gestión de Información de Profesores y Estudiantes de la Facultad 6*. Esta aplicación está delimitada en seis módulos, dedicados cada uno a agilizar y facilitar la manipulación de dicha información.

Ellos son:

- ✓ **Módulo Profesores:** empleado para gestionar la información de los profesores de la facultad durante su estancia en la misma.
- ✓ **Módulo Estudiantes:** creado para gestionar la información de los estudiantes de la facultad en el transcurso de los cursos académicos.
- ✓ **Módulo Residencia:** empleado para manipular toda la información referente a las ubicaciones de los estudiantes en la beca así como la información de los apartamentos con que cuenta la facultad.
- ✓ **Módulo Extensión:** utilizado para gestionar la información referente a las actividades extensionistas en las que participan tanto los estudiantes como los profesores de la facultad.
- ✓ **Módulo Producción:** usado para gestionar la información relacionada con la producción en la facultad, dígase los proyectos productivos, los polos a los que pertenecen los mismos, los laboratorios de producción, etc.
- ✓ **Módulo Tesis:** diseñado para controlar la información relacionada con las tesis, estudiantes, tutores y oponentes de la facultad.

La información con la que se trabaja en la aplicación se encuentra limitada en cada uno de los seis módulos y de manera independiente. Esto imposibilita el flujo de información entre los módulos y en algunos casos ésta se encuentra duplicada al estar presente en varios módulos a la vez. Debido a la necesidad que existe de trabajar los datos de forma centralizada y de garantizar su flujo entre los módulos para así evitar la redundancia y repetición de los mismos, surge la necesidad de diseñar e implementar una arquitectura que soporte la integración de los módulos en una aplicación común para todos.

Por todo lo antes planteado se define como **Problema Científico**:

¿Cómo contribuir a la integración eficiente de los módulos del Sistema para la Gestión de la Información de Profesores y Estudiantes de la Facultad 6?

Con vista a la solución del problema científico se plantea como **Objeto de Estudio**:
El proceso de desarrollo de software para sistemas de gestión de información.

A partir del objeto de estudio se delimita el siguiente **Campo de Acción**:
La descripción y diseño de la arquitectura para sistemas de gestión de información de la Facultad 6.

Se persigue como **Objetivo General**:
Definir la arquitectura del Sistema para la Gestión de la Información de Profesores y Estudiantes de la Facultad 6.

Para cumplir este objetivo general se trazaron los siguientes **Objetivos Específicos**:

1. Identificar patrones y estilos arquitectónicos.
2. Definir herramientas y tecnologías.
3. Elaborar el documento de arquitectura de software.
4. Validar la arquitectura.
5. Integrar la aplicación haciendo uso de la arquitectura validada.

Se desarrollarán las siguientes **Tareas** para dar cumplimiento a los objetivos trazados:

1. Estudio de la arquitectura de los sistemas de gestión de información similares.
2. Estudio de los diferentes patrones y estilos arquitectónicos.
3. Fundamento de las herramientas y tecnologías a utilizar.
4. Creación de la vista lógica.
5. Creación de la vista de procesos.
6. Creación de la vista de casos de uso.
7. Creación de la vista de implementación.
8. Creación de la vista despliegue.
9. Construcción y evaluación de la prueba de concepto de arquitectura.
10. Definición de la sincronización y actualización de la base de datos.
11. Definición de los estilos *CCS* a utilizar en el diseño de la aplicación.

CAPÍTULO 1: Fundamentación teórica

Introducción

En este capítulo se hace referencia a los conceptos vinculados al objeto de estudio, los tipos de arquitectura y los patrones arquitectónicos más utilizados en el mundo y los definidos para utilizar en el desarrollo del *Sistema para la Gestión de Información de Profesores y Estudiantes de la Facultad 6*. Se realiza un estudio de los lenguajes, tecnologías y herramientas existentes.

1.1 Arquitectura de Software

La arquitectura del software puede ser referenciada como el diseño de más alto nivel de la estructura de un sistema informático. La arquitectura de software, también denominada arquitectura lógica, no es más que el conjunto de patrones y abstracciones coherentes que proporcionarán el marco de referencia necesario para guiar la construcción del software para el *Sistema para la Gestión de Información de Profesores y Estudiantes de la Facultad 6*. Ésta establece además, los fundamentos para que los analistas, diseñadores y programadores que integran el equipo de desarrollo trabajen en una línea común (línea base de la arquitectura) que permita alcanzar los objetivos que se proponen, cubriendo todas las necesidades y especificidades planteadas por los clientes.

La arquitectura de software se selecciona y diseña con base en los objetivos prefijados para el desarrollo del sistema, pero no solamente los de tipo funcional, también otros objetivos de tipo no funcionales tales como la mantenibilidad, seguridad, portabilidad, modificabilidad y flexibilidad. También la arquitectura tiene bases en las restricciones o limitaciones derivadas de las tecnologías disponibles para llevar a cabo la implementación del sistema.

Definición de Arquitectura de Software

En la actualidad no existe, debido a su flexibilidad, una definición única para la arquitectura de software. Algunas de las definiciones más comunes y que se tienen en cuenta por los arquitectos son:

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

Kruchten, Philippe [1].

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

Paul Clements (Clements, 1996) [2].

La arquitectura de software es la organización fundamental de un sistema encarnada en sus componentes, en sus relaciones mutuas y en el entorno, y los principios que guían su diseño y evolución.

Standard IEEE 1471-2000 [3].

A pesar de la diversidad de definiciones existentes, se pueden notar elementos comunes entre las mismas, ya que la arquitectura de software define, de manera abstracta, los componentes que llevan a cabo alguna tarea de computación, sus interfaces y la comunicación entre ellos y con el entorno en el cual se encuentra. Toda arquitectura debe ser implementable en una arquitectura física, que consiste simplemente en determinar qué nodo de procesamiento tendrá asignada cada tarea.

1.2 Estilos y patrones

1.2.1 Estilos arquitectónicos

Los estilos definen una familia de sistemas en términos de un patrón de organización estructural. Un estilo define además el vocabulario de tipos de componentes y conectores, las restricciones de combinación que se establecen y uno o más modelos semánticos para determinar propiedades del todo a partir de las partes.

Los estilos arquitectónicos agrupan una serie de elementos (*Componentes, Conectores, Configuraciones y Restricciones*) que permiten conformar una descripción con alto nivel de abstracción la cual puede realizarse en lenguaje natural, pero se recomienda que sea mediante los *Lenguajes de Descripción Arquitectónica (ADLs)* [11].

Los estilos expresan la arquitectura de un sistema en el sentido más formal y teórico. Describen una clase de arquitectura o piezas identificables de las arquitecturas que se encuentran repetidamente en la práctica. Es por eso que una vez identificados se puede pensar en re-utilizarlos en situaciones semejantes que se presenten posteriormente. Igual que los patrones de arquitectura y de diseño, todos los estilos tienen un nombre: cliente-servidor, modelo-vista-controlador, tubería-filtros, arquitectura en capas, etc.

Son diversos los tipos de estilos y diversas también las formas de agruparlos en dependencia del comportamiento de los sistemas en los que se emplean. Para la arquitectura de SIGIPE se pretende adoptar el *Modelo Vista Controlador (MVC)* de la familia de los estilos basados en llamada y retorno. Este estilo pertenece a la familia de los que enfatizan la modificabilidad y escalabilidad y fue reconocido como un estilo por *Taylor* y *Medvidovic* [4] y en ocasiones se define como un patrón de diseño o como una práctica recurrente en la ingeniería de software.

1.2.2 Patrones de Software

Patrones arquitectónicos

Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, para describir después el núcleo de la solución a ese problema, de tal manera que esa solución pueda ser usada más de un millón de veces sin hacerlo ni siquiera dos veces de la misma forma.

Christopher Alexander [5].

Los patrones conforman una práctica de diseño que se origina previamente a la distinción de la arquitectura de software como discurso en perpetuo estado de formación y proclamara su independencia de la ingeniería en general y el modelado en particular.

Billy Reinoso, 2004 [6].

Teniendo en cuenta que en un marco de arquitectura tanto los estilos como los patrones arquitectónicos suelen ser referenciados o incluso usados indistintamente, algunos arquitectos experimentados asumen que los patrones arquitectónicos ascienden a un alto nivel de diseño alejándose de la teoría y acercándose más a la implementación.

A grandes rasgos, se puede decir que un patrón arquitectónico se compone de tres elementos fundamentales:

- ✓ **El contexto**, como situación de diseño en la que surge un problema de diseño.
- ✓ **El problema,** como el conjunto de fuerzas que aparecen repetidamente en el contexto.
- ✓ **La solución**, la cual surge como la configuración destinada a equilibrar o estabilizar las fuerzas del problema y que abarcan, en su aplicación, una estructura con componentes y sus relaciones y un comportamiento en tiempo de ejecución que refleja los aspectos dinámicos de la solución como la colaboración entre componentes y la comunicación entre ellos.

En lo que a patrones de arquitectura se refiere, se plantea que existen dos tendencias básicas de los mismos.

Primeramente tenemos a los *Patrones Orientados a Arquitectura de Software (POSA)* los cuales serán los usados para el desarrollo del sistema *SIGIPE*, entre varias razones porque agrupan una vista general de distintos niveles de abstracción en concepción de patrones, entre ellos un grupo orientado a un alto nivel de arquitectura y entre los mismos se encuentra el *Modelo-Vista-Controlador,* el cual es empleado por el framework de desarrollo *Symfony*, definido para el desarrollo de la aplicación.
Los siguientes, también tratados indistintamente como estilos, son algunos de estos patrones:

- ✓ Layers
- ✓ Pipes and Filters
- ✓ Presentation-Abstraction-Control
- ✓ Microkernel
- ✓ Reflection
- ✓ Model-View-Controller

Este último divide una aplicación interactiva en tres componentes fundamentales:
El *Modelo*, que contiene los datos y funcionalidades de fondo; la *Vista*, exhibe información para el usuario y por último el *Controlador*, encargado de manipular las entradas del usuario.

La *Vista* y el *Controlador* conjuntamente conforman la interfaz que interactúa directamente con el usuario. El *Controlador* supone además un mecanismo de propagación de cambio que asegura la consistencia necesaria entre la interfaz de usuario y el modelo, para el correcto flujo de comunicación entre los componentes del patrón.

Por otra parte tenemos a los *Patrones de Arquitectura Empresarial (PAE)*. Estos patrones se centran básicamente en unificar las partes de una aplicación empresarial mediante formas comunes por lo que no serán usados para el desarrollo del sistema.

Patrones de diseño

Dentro del marco de la arquitectura de software se insertan lógicamente los elementos relacionados al diseño del sistema en cuestión, en este caso, una aplicación para gestión de información. De manera similar a los patrones de arquitectura, también existen los patrones de diseño. Los mismos tienen a su cargo la definición de cuatro elementos fundamentales aplicados al modelo de clases del sistema.
Dichos elementos son:

- ✓ **Nombre:** describe el problema de diseño, su solución, y consecuencias en una o dos palabras.
- ✓ **Problema:** describe cuándo aplicar el patrón. Se explica el problema y su contexto. Puede describir estructuras de clases u objetos que son sintomáticas de un diseño inflexible.
- ✓ **Solución:** describe los elementos que forman el diseño, sus relaciones, responsabilidades y colaboraciones. No se describe un diseño particular ya que un patrón es una plantilla que puede ser reutilizable siempre que sea posible.
- ✓ **Consecuencias:** resultados de aplicar el patrón.

Un patrón de diseño, obviamente, debe encajar por un lado con otros tipos de patrones y por el otro con la teoría, la práctica y los marcos que en general rigen el diseño. En el presente documento se propone realizar un diseño arquitectónico basado en casos de uso como modelo de las funciones que debe cumplir la aplicación. Sirviendo además como un contrato o acuerdo entre el cliente y los desarrolladores. Así se posibilita describir el proceso en términos de actividades, trabajadores y artefactos, definiéndose además la organización del mismo en las fases y flujos de trabajo aportados por la metodología *RUP (Racional Unified Process)* propuesta para el desarrollo de la aplicación.

Dado al hecho de que para el desarrollo de la aplicación se hará uso del framework de desarrollo *Symfony*, así como de sus patrones de software, la arquitectura de la aplicación estará moldeada por los mismos, es decir, que la arquitectura de la aplicación estará regida en cierta medida por la arquitectura propuesta por el framework.

Algunos de los patrones empleados por el framework y que también se agrupan según sus características, son los que a continuación se relacionan.

Los *Patrones de Software para Asignación General de Responsabilidades (GRASP: General Responsability Assignment Software Patterns)*:

- ✓ **Creador:** En las clases *Actions* se encuentran las acciones definidas para nuestra aplicación. En estas se crean los objetos de las clases que representan las entidades de la base de datos, por lo que de este modo las clases *Actions* son las "creadoras" de dichas entidades.
- ✓ **Experto:** Este es uno de los más utilizados, puesto que *Propel* es la librería externa que utiliza *Symfony* para realizar su capa de abstracción en el modelo. El mismo encapsula toda la lógica de los datos y son generadas las clases con todas las funcionalidades comunes de las entidades.
- ✓ **Alta Cohesión:** *Symfony* permite asignar responsabilidades con una alta cohesión, por ejemplo las clases *Actions* tiene la responsabilidad de definir las acciones para las plantillas y colaboran con otras para realizar diferentes operaciones, instanciar objetos y acceder a las *Properties*, es decir, están formadas por diferentes funcionalidades que se encuentran estrechamente relacionadas garantizando que el software sea flexible frente a grandes cambios.
- ✓ **Controlador:** Todas las peticiones Web son manejadas por un solo controlador frontal *(sfActions)*, que es el único punto de entrada de la aplicación en un entorno determinado. Cuando el controlador frontal recibe una petición, utiliza el sistema de enrutamiento para asociar el nombre de una acción y el nombre de un módulo con la URL entrada por el usuario.
- ✓ **Bajo Acoplamiento:** La clase *Action* hereda solamente de *sfActions* para lograr un bajo acoplamiento de clases.

Los patrones *GoF (Gang of Four)*:

- ✓ **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. En el controlador frontal hay una llamada a *sfContext::getInstance()*. En una acción, el método *getContext()*, un objeto muy útil que guarda una referencia a todos los objetos del núcleo de *Symfony*.
- ✓ **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. Cuando el framework necesita por ejemplo crear un nuevo objeto para una petición, busca en la definición de la factoría el nombre de la clase que se debe utilizar para esta tarea.

Siendo estos últimos de la categoría de los creacionales.

Symfony también está basado en el clásico patrón de diseño web conocido como patrón de diseño *Modelo-Vista-Controlador (MVC).* El mismo es usado además de patrón arquitectónico como patrón de diseño separando el modelado del dominio, la presentación y las acciones basadas en datos ingresados por el usuario en tres clases diferentes como se mencionó anteriormente.

Estas clases son:

- **Modelo:** El modelo administra el comportamiento y los datos del dominio de la aplicación, responde a requerimientos de información sobre su estado (usualmente formulados desde la *Vista*) y responde a instrucciones de cambiar el estado (habitualmente desde el *Controlador*).
- **Vista:** Maneja la visualización de la información e interactúa directamente con los usuarios.
- **Controlador:** Interpreta las acciones del ratón y el teclado, informando al modelo y/o a la *Vista* para que cambien según resulte apropiado.

Tanto la *Vista* como el *Controlador* dependen del modelo, el cual no depende de las otras clases y permite construir y probar el *Modelo* independientemente de la representación visual.

Entre las ventajas del estilo señaladas en la documentación de *Patterns & Practices* de *Microsoft* están las siguientes [7]:

- ✓ **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 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. Los usuarios pueden preferir distintas opciones de representación, o requerir soporte para nuevos dispositivos como teléfonos celulares o *PDAs (Personal Digital Assistant o Asistente Personal Digital)*. Dado que el *Modelo* no depende de las *Vistas*, agregar nuevas opciones de presentación generalmente no afecta al *Modelo*. Este patrón sentó las bases para especializaciones ulteriores, tales como *Page Controller (Controlador de Página)* y *Front Controller (Controlador Frontal)*.

Entre las desventajas, se han señalado [7]:

- ✓ **Complejidad:** El patrón profundiza la orientación a eventos del código de la interfaz de usuario, que puede llegar a ser difícil de depurar. En rigor, la configuración basada en eventos de dicha interfaz corresponde a un estilo particular (arquitectura basada en eventos) que aquí se examina por separado.
- ✓ **Costo de actualizaciones frecuentes:** Desacoplar el *Modelo* de la *Vista* no significa que los desarrolladores del *Modelo* puedan ignorar la naturaleza de las *Vistas*. Si el *Modelo* experimenta cambios frecuentes, podría desbordar las *Vistas* con una lluvia de requerimientos de actualización. Las *Vistas* de las pantallas gráficas involucraban más tiempo para plasmar el dibujo que el que demandaban los nuevos requerimientos de actualización.

Es necesario acotar que aún cuando se ha hecho el señalamiento de estas desventajas, las mismas no entorpecen la utilización de dicho patrón para el desarrollo de la aplicación y aún así, sigue siendo el mismo, uno de los más utilizados en el mundo en el desarrollo de aplicaciones.

1.3 Lenguajes de Descripción Arquitectónica

Los lenguajes de descripción de arquitecturas, o ADLs, ocupan una parte importante del trabajo arquitectónico. Los mismos constituyen una alternativa a las herramientas que no cumplen con la satisfacción de requerimientos descriptivos de alto nivel de abstracción, en otras palabras, no se prestan a las mismas tareas que los ADLs diseñados específicamente para esa finalidad. Los ADLs permiten modelar una arquitectura previa a la programación de las aplicaciones que la componen, analizar su adecuación, determinar sus puntos críticos y eventualmente simular su comportamiento [11].

En la actualidad no existe una definición única de ADL, pero comúnmente se acepta que un ADL debe proporcionar un modelo explícito de componentes, conectores y sus respectivas configuraciones. Se estima deseable, además, que un ADL suministre soporte de herramientas para el desarrollo de soluciones basadas en arquitectura y su posterior evolución. Contando con un ADLs, un arquitecto puede razonar sobre las propiedades del sistema con precisión, pero a un nivel de abstracción convenientemente genérico.

Billy Reinoso, 2004 [6].

La definición más simple que define un ADL como una entidad consistente en cuatro "Cs":
Componentes
Conectores
Configuraciones
Restricciones (Constraints).

Tracz [Wolf97] [8].

Según Steve Vestal (1993) [12], y cito: *La definición del ADLs que habrá de aplicarse en lo sucesivo es la de un lenguaje descriptivo de modelado que se focaliza en la estructura de alto nivel de la aplicación antes que en los detalles de implementación de sus módulos concretos.*

Entre los *ADLs* más destacados y usados actualmente se encuentran:

- ✓ Acme-Armani (Acme es un lenguaje de intercambio de ADLs, Armani es un ADL asociado a Acme)
- ✓ Aesop (ADL de propósito general con énfasis en estilos)
- ✓ xADL (ADL basado en XML)

- ✓ SADL (ADL con énfasis en mapeo de refinamiento)
- ✓ UML (Lenguaje genérico de modelado)

El *UML* es un caso muy particular ya que aunque forma parte del repertorio de lenguajes de modelado, un gran número de arquitectos no lo admiten como un *ADL*. Actualmente existen dos tendencias sobre el uso de *UML* como *ADL*, en una se impulsa su uso casi irrestricto como tal y en la segunda se señalan las limitaciones de *UML* no sólo como *ADL* sino como lenguaje universal de modelado.

Ya se ha dicho que *UML* no es en modo alguno un *ADL* y que los manuales contemporáneos carecen del concepto de estilo además de la falta de relación entre sus definiciones de arquitectura con lo que la misma significa en el campo de los *ADLs*. Sin embargo, para la descripción arquitectónica de SIGIPE se propone el uso de *UML*, debido al hecho de que constituye una herramienta de uso habitual en el modelado y que una gran cantidad de importantes autoridades en *ADLs*, siguiendo a *Nenad Medvidovic*, han indicado la posibilidad de utilizarlo como metalenguaje para implementar la semántica de dos *ADLs (SADL y Wright)*. También debido a que el meta-modelo de *UML* puede constituir un equivalente a la ontología arquitectónica de *Acme*.

1.4 Lenguaje Unificado de Modelado (UML)

El *Lenguaje Unificado de Modelado (UML)* es un lenguaje de modelado visual que se usa para especificar, visualizar, construir y documentar artefactos de un sistema de software. Aún cuando se reconoce que *UML* no cuenta con un nivel de abstracción ideal para describir una arquitectura, es hoy el lenguaje más utilizado para llevar a cabo esta tarea.

Citando a *Schmuller (2000)*, es sumamente importante destacar que un modelo *UML* describe lo que supuestamente hará un sistema, pero no dice cómo implementar dicho sistema.

El modelo más aceptado a la hora de establecer las vistas necesarias para describir una arquitectura de software es el modelo 4+1 vistas arquitectónicas. Aunque no existe de forma explícita una vista arquitectónica, estas cinco vistas pretenden describir en su conjunto la arquitectura del sistema.

Kruchten, 1995 [9].

Las cuatro vistas principales definidas en este modelo son:

- ✓ **Vista Lógica:** modelo de objetos, clases, entidad – relación.
- ✓ **Vista de Procesos:** modelo de concurrencia y sincronización.
- ✓ **Vista de Desarrollo:** organización estática del software en su entorno de desarrollo.
- ✓ **Vista Física:** modelo de correspondencia software – hardware.

Además de estas cuatro vistas principales, existe otra vista, la "+1". La misma se muestra y traza en cada una de las cuatro principales y está formada por las necesidades funcionales que cubre el sistema, a partir de lo cual es identificada como *Vista de Casos de Uso*. A partir de ello se deduce que, según este modelo, la arquitectura es en realidad evolucionada desde los escenarios de los *Casos de Uso*, es decir, que la arquitectura es dirigida por los casos de uso.

Según la metodología de *RUP*, estas vistas se definen:

- ✓ **Vista Lógica:** Muestra elementos de diseño arquitectónicamente significativos del sistema. Soporta los requerimientos funcionales, identifica mecanismos y diseña elementos comunes a través del sistema.

- ✓ **Vista de Procesos:** Muestra la estructura de procesos del sistema. Especifica las líneas de mando que ejecutan cada operación en cada una de las clases señaladas en la vista lógica.
- ✓ **Vista de Implementación:** Agrupa una colección de componentes y subsistemas de implementación que proveen las funcionalidades del sistema.
- ✓ **Vista de Despliegue:** Cuenta con una descripción de los nodos físicos de los cuales se constituye el sistema.
- ✓ **Vista de Casos de Uso:** Modela los casos de uso arquitectónicamente significativos del sistema. Aquellos necesarios para el usuario final. Además actúa como indicador que ayuda al diseñador a descubrir los elementos de la arquitectura durante su diseño y valida e ilustra el diseño de la misma.

Algunas de las características que propician que con el uso de *UML* se pueda desarrollar un modelado eficiente son la simplicidad de la comunicación entre desarrolladores de software, la facilidad de entendimiento y aprendizaje de sus principios principales, permite y viabiliza la comunicación entre trabajadores del proyecto y usuarios, la estandarización de los elementos del diseño de sistemas y que constituye el estándar más utilizado mundialmente.

1.5 Metodología de desarrollo de software

La creciente informatización de los procesos productivos y sociales ha traído consigo que las organizaciones y empresas requieran cada vez más de software confiable y de alta calidad, tanto en su desarrollo como en su mantenimiento. Es por ello que en los últimos años se han venido publicando estándares, notaciones y metodologías que establecen buenas prácticas para los procesos de desarrollo de software.

Una metodología de desarrollo como conjunto de procedimientos, técnicas, herramientas, y soporte documental guía a los desarrolladores durante el ciclo de desarrollo de un software. En otras palabras, una metodología representa el camino para desarrollar software de una manera sistemática, persiguiendo tres necesidades principales:

- ✓ Mejores aplicaciones, conducentes a una mejor calidad.
- ✓ Un proceso de desarrollo controlado.
- ✓ Un proceso normalizado en una organización, no dependiente del personal.

Algunas de las metodologías que se suelen utilizar para guiar un proceso de desarrollo de software son la metodología *Rational Unified Process (RUP)*, la cual es más adaptable para proyectos de largo plazo, la metodología *Extreme Programming (XP)*, la cual por su parte se recomienda para proyectos de corto plazo y la metodología *Microsoft Solution Framework (MSF)*, la cual se adapta a proyectos de cualquier dimensión y de cualquier tecnología.

En ocasiones, el diseño de un software se realiza de manera rígida con los requerimientos que el cliente solicitó, provocando un difícil mantenimiento si el usuario final solicita un cambio durante la fase de prueba. Esto se debe a no usar o aplicar una metodología adecuada generalmente debido a la falta de conocimiento sobre las mismas.

Por su parte la metodología *XP* no es conveniente que sea usada ya que esta se basa en el trabajo orientado directamente al objetivo. Para esto se apoya en las relaciones interpersonales y en la velocidad de reacción para la implementación y para los cambios que puedan surgir durante el desarrollo del

proceso. Aunque de cierto modo se logra minimizar el riesgo de fallo en el proceso, se necesita de un representante del cliente dentro del equipo de desarrollo, que se encargue de responder las preguntas y dudas que surjan por parte del equipo de desarrollo durante el proceso y para el desarrollo del proyecto no se cuenta con dicho representante. Otra de las características por las que no se decide el uso de XP, es que se centra en *Historias de Uso (UseStories)* las cuales son descritas por el cliente o su representante en el equipo de desarrollo. Además, el código siempre se produce en parejas que van cambiando constantemente para lograr así que todo el equipo sepa y pueda modificar según necesidades el código generado. Esto último no se ajusta a las necesidades del proyecto, ya que los desarrolladores de los módulos no rotarán posiciones para intervenir en el desarrollo de otro módulo.

En el caso de *Microsoft Framework Solutions (MSF)* es válido aclarar que aporta grandes ventajas durante el desarrollo de un software. Es una metodología ajustable a proyectos de pequeña y gran escala incentivando al trabajo en equipo y la colaboración. *MSF* como metodología al fin desde cierto punto de vista aporta ventajas para el desarrollo de productos de software, pero también presenta algunas dificultades que propiciaron esquivar esta metodología para el desarrollo del proyecto. Por ser un modelo prescriptivo requiere de mucha documentación en cada fase y el necesario análisis de riesgos puede demorar o incluso frenar el avance del proyecto. Otra de las desventajas que presenta esta metodología es que por estar basada en tecnología *Microsoft* no presenta mucha flexibilidad en cuanto al uso de herramientas y trata de forzar al equipo de desarrollo al uso de solo herramientas propietarias de Microsoft.

En el caso del *Sistema para la Gestión de Información de Profesores y Estudiantes de la Facultad 6* se determina adoptar la metodología *RUP* para guiar el desarrollo de dicho proyecto ya que al realizar una comparación con la metodología *XP*, se pudo observar que esta última no es la más adecuada para este proyecto, ya que *XP* requiere que el cliente forme parte del grupo de desarrollo y ésta es una característica que no se ajusta a lo que se está buscando. Otro motivo por el cual no se decidió adoptar *XP* como metodología es que la misma por tratar de agilizar la entrega pone en riesgo la solidez de la arquitectura desde el punto de vista que lo que hace es buscar un avance y mostrárselo al cliente y repite estas acciones constantemente. Es rápido, pero se olvida un poco de lo importante que es una buena

arquitectura. *RUP*, a diferencia, es ajustable y permite definir menos actividades y menos documentación, de manera tal que puedes lograr un proceso más rápido.

En el caso de *RUP*, está muy bien documentado y define roles fundamentales que tienen a su cargo la generación de artefactos y documentación concretos por cada fase y flujo que tributan a un buen producto final. Además es una de las más usadas mundialmente y esto se debe a debido a los resultados que se obtienen a partir de su aplicación.
RUP es una metodología que centra su ciclo de vida en la arquitectura y está diseñada de forma que exista un entendimiento continuo y gradual de todos los implicados en el proyecto, dígase tanto clientes como desarrolladores. Otra de las particularidades de esta metodología que da un punto a favor de su uso, es que en cada ciclo de iteración se hace exigente el uso de artefactos, siendo por este motivo, una de las metodologías más importantes para alcanzar un grado de certificación en el desarrollo del software.

La metodología *RUP* divide en 4 fases el proceso de desarrollo de software:

1- **Conceptualización (Concepción o Inicio):** Se describe el negocio y se delimita el proyecto describiendo sus alcances con la identificación de los casos de uso del sistema.

2- **Elaboración:** Se define la arquitectura del sistema y se obtiene una aplicación ejecutable que responde a los casos de uso que la comprometen. A pesar de que se desarrolla a profundidad una parte del sistema, las decisiones sobre la arquitectura se hacen sobre la base de la comprensión del sistema completo y los requerimientos (funcionales y no funcionales) identificados de acuerdo al alcance definido.

3- **Construcción:** Se obtiene un producto listo para su utilización que está documentado y tiene un manual de usuario. Se obtiene una o varias liberaciones del producto que han pasado las pruebas. Se ponen estas entregas a consideración de un subconjunto de usuarios.

4- **Transición:** El producto ya está listo para su instalación en las condiciones reales. Puede implicar reparación de errores.

Cada una de estas fases es desarrollada mediante ciclos de iteraciones.

Cada ciclo de vida que se desarrolla por iteraciones es llevado bajo dos disciplinas o flujos de trabajo.

Disciplina de desarrollo o flujos de ingeniería:

- ✓ **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 un software que se ajuste a la arquitectura y que tenga el comportamiento deseado.
- ✓ **Pruebas:** Búsqueda de errores durante el ciclo de vida del proyecto.
- ✓ **Despliegue:** Se desarrollan actividades de empaquetamiento e instalación para la entrega.

Disciplina o flujos de soporte:

- ✓ **Configuración y administración de cambios:** Guardando todas las versiones del proyecto.
- ✓ **Administración del proyecto:** Administrando horarios y recursos disponibles.
- ✓ **Ambiente:** Administrando el ambiente de desarrollo.
- ✓ **Distribución:** Hacer todo lo necesario para la salida del proyecto.

1.6 Lenguajes, tecnologías, herramientas de soporte al desarrollo y herramientas CASE

Con la idea de mejorar la facilidad y calidad de trabajo en todas partes del mundo se han creado un gran número de lenguajes de programación. Muchos de estos lenguajes son utilizados para el desarrollo de aplicaciones para la automatización de tareas, permitiendo a las personas trabajar con mayor comodidad y obtener mejores resultados. Dentro del conjunto de aplicaciones que se pueden crear mediante el uso de los lenguajes están las aplicaciones Web, las cuales son desarrolladas mediante el uso de varios lenguajes como *HTML (Hypertext Markup Language)*, *DHTLM (Dynamic HTML)*, *XML (Extensible Markup Language)*, *JavaScript*, *PHP (Personal Home Page)*.

Revisando el conjunto de lenguajes disponibles para desarrollo Web, existen muchos de los mismos que son sólo para la programación del lado del cliente, imposibilitando la ejecución de código en el servidor. Por otra parte, muchos de los que permiten la programación del lado del servidor no son gratis, no proporcionan la robustez requerida en las aplicaciones o carecen de ciertas ventajas claves como la *Programación Orientada a Objetos (POO)*. Por tales motivos se propone que el lenguaje de programación a utilizar para la implementación de la aplicación sea *PHP* en su versión 5.2.6 ya que una vez comparado con *Java* se pudo apreciar que *Java* está pensado para proyectos más grandes, más robustos, más complejos y partiendo que lo que se quiere es un sistema para gestionar información, este debe de ser ágil y para eso con Java no se puede contar. La ventaja de *PHP* en su versión 5.2.6 consiste en la programación orientada a objetos, donde ha hecho mejoras de los mecanismos de utilización de dicho recurso para solucionar las carencias de las versiones anteriores. Es un lenguaje libre y soportado por un gran número de frameworks como *Zend*, *Seagull*, *Prado*, *Codeigniter* y *Symfony*. Éstos últimos han surgido como estructuras de soporte definidas en las cuales otros proyectos de software pueden ser organizados y desarrollados sin la necesidad de tener que comenzar desde cero.

También se utilizará el *IDE (Integrated Development Environment) ECLIPSE* 3.4, el cual brinda nuevas facilidades como la disposición de un entorno mucho más flexible y profesional. Además, brinda refactorización del código fuente, permitiendo adecuar el comportamiento externo de una función o clase sin cambiar el funcionamiento interno. Es un *IDE* que cuenta con compilación en tiempo real y permite llevar a cabo el control de versiones con *Subversion* a través de plugins libremente disponibles. Su fácil

integración con *Subversion* y su buen completamiento de código son elementos claves para seleccionarlo como entorno de desarrollo.

Además para tener un control sobre las versiones de la aplicación se precisa la utilización del *Subversion* 1.4.5 para *Apache* 2.2. Un software de sistema de control de versiones diseñado específicamente para reemplazar al popular *Current Version System (CVS)*, el cual posee varias deficiencias. El *Subversion* es software libre bajo una licencia de tipo *Apache/BSD* y se le conoce también como *SVN* por ser ese el nombre de la herramienta de línea de comandos. Una característica importante de *Subversion* es que, a diferencia de *CVS*, los archivos versionados no tienen cada uno un número de revisión independiente. En cambio, todo el repositorio tiene un único número de versión que identifica un estado común de todos los archivos del repositorio en cierto punto del tiempo.

Entre las principales ventajas que propician el uso de este software tenemos:

- ✓ La creación de ramas y etiquetas es una operación más eficiente; Tiene costo de complejidad constante (O(1)) y no lineal (O(n)) como en *CVS*.
- ✓ Permite que sólo se envíen las diferencias en ambas direcciones (en *CVS* siempre se envían al servidor archivos completos).
- ✓ Permite selectivamente el bloqueo de archivos. Se usa en archivos binarios que, al no poder fusionarse fácilmente, conviene que no sean editados por más de una persona a la vez.
- ✓ Cuando se usa integrado a Apache permite utilizar todas las opciones que este servidor provee a la hora de autentificar archivos (*SQL, LDAP, PAM*, etc.).

Para soportar el ciclo de vida del desarrollo del proyecto se hará uso de la herramienta *Visual Paradigm* para *UML versión 6.1*. La misma permite realizar 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 costo. 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*. Otra de las características de *Visual Paradigm* para *UML* es que aporta a los desarrolladores de software una plataforma de desarrollo para construir aplicaciones de calidad con gran

rapidez. Aporta además, una excelente interoperabilidad con otras herramientas *CASE* y muchos de los entornos *IDE* más utilizados hoy día.

Algunas de las características tomadas en cuenta para su uso durante el desarrollo del proyecto son:

- ✓ Ofrece entorno de creación de diagramas para *UML 2.0*.
- ✓ Disponibilidad en múltiples plataformas.
- ✓ Disponibilidad de integrarse en los principales *IDEs*.
- ✓ Soporta una gama de lenguajes para la generación de código *Java*, *C++*, *PHP*, *XML*, *Python*, etc.
- ✓ La generación de código soporta *C #*, *VB*, *.NET*, *C++*, *Delphi*, *Perl*, y *Ruby* entre otros.

1.6.1 Plataforma y Framework.

La plataforma:

Partiendo de que la aplicación será implementada con *PHP* y que el gestor de base de datos será *PostgreSQL*, ambos disponibles tanto para la plataforma *Windows* como para Linux, el sistema podrá ser utilizado en cualquiera de estas dos plataformas aunque será desarrollado y desplegado sobre *Windows*.

El framework:

Hoy día el uso de los frameworks se ha popularizado a gran escala, facilitando a los desarrolladores la creación de aplicaciones con un coste de tiempo considerablemente menor que hace unos años cuando éstas no estaban presente. El número de frameworks de desarrollo con el que se cuenta actualmente es relativamente elevado y supera los veinte. Entre los más usados se encuentran *Zend*, *Prado*, *CakePHP*, *Codeigniter*, *Solar*, *PHPOnTrax* y *Symfony*. En general los frameworks ofrecen diversas ventajas como la disponibilidad de documentación suficiente, posibilitando trabajar con los gestores de bases de datos más conocidos, además ofrecen patrones de programación, estructura, desarrollo y mantenimiento del código y tienen la capacidad de separar la funcionalidad, los datos y la presentación.

El framework *Symfony* en la versión 1.2.2 fue seleccionado para el desarrollo de la aplicación *SIGIPE* puesto que se ajusta a las necesidades del equipo de desarrollo en cuanto a implementación. Este es un completo framework que permitirá optimizar el desarrollo de la aplicación, además, proporcionará varias herramientas y clases encaminadas a reducir el tiempo de desarrollo de la aplicación por compleja o sencilla que sea. También automatizará las tareas más comunes, permitiendo a los desarrolladores dedicarse por completo a los aspectos específicos de la aplicación. *Symfony* está desarrollado completamente con PHP 5, ha sido probado en numerosos proyectos reales y se utiliza en sitios Web de comercio electrónico de primer nivel. Además es compatible con la mayoría de gestores de bases de datos como *MySQL*, *PostgreSQL*, *Oracle* y *Microsoft SQL Server*, siendo independiente de los mismos. Se puede ejecutar tanto en plataformas *nix (Unix, Linux, etc.) como en plataformas *Windows* con la garantía de que funciona correctamente en éstos sistemas. *Symfony* en comparación con otros frameworks es más eficiente y logra que el equipo de desarrollo trabaje más organizado y que se centre solamente en la programación en *PHP* ya que posee desde *helpers* para facilitar la programación del diseño hasta clases y librerías para la abstracción del gestor utilizado. Otra ventaja de *Symfony* con

respecto a los otros frameworks ya mencionados es que utiliza varios patrones garantizando una arquitectura consistente y robusta para el sistema.

En el caso particular de *Symfony* en la versión 1.2.2, es una versión estable que presenta algunas mejoras sobre versiones anteriores que benefician el desarrollo del sistema. Por ejemplo en esta versión *Propel* fue mejorado a la versión 1.3, en el caso de los formularios, estos presentan mejoras ya que se puede verificar si el formulario contiene errores en alguno de los campos con sólo usar `sfForm::hasErrors()`. Por otra parte cuando se guarda un formulario *Symfony* automáticamente guarda el formulario principal y los objetos relacionados con los formularios embebidos, característica ausente en versiones anteriores. Esta versión también presenta mejoras en los validadores, por ejemplo, ahora permite usar el operador "**==**" en vez de "sfValidatorSchemaCompare::EQUAL" para realizar comparaciones. *Symfony* 1.2.2 aporta nuevos métodos para validar rangos de fechas como es el caso de "`sfValidatorDateRange`", muy necesarios para el sistema que se va a implementar puesto que se trabaja con un gran número de fechas. Además de esto, los *widgets* pueden definir sus propios estilos. Si en el momento en que se está generando el modelo ocurre algún fallo, el framework muestra un mensaje de error más claro acerca del error.
Estos entre otros muchos beneficios que aporta esta versión respecto a las anteriores permiten que esta sea la versión de este framework para utilizar en el desarrollo del sistema.

1.6.2 Gestor de Base de datos

El sistema gestor de bases de datos que almacena los datos de la aplicación es *PostgreSQL v8.2*.
PostgreSQL es un sistema de base de datos relacional perteneciente al ámbito del software libre que destaca por su robustez, escalabilidad y cumplimiento de los estándares *SQL*. Cuenta con versiones para una amplia gama de sistemas operativos, entre ellos: *Linux, Windows, Mac SX, Solaris, BSD, Tru64* y otros más.

PostgreSQL soporta *ACID (Atomicity, Consistency, Isolation and Durability), o lo qu*e es lo mismo, la realización de transacciones seguras, también *Vistas*, *Uniones*, *Claves extranjeras*, *Procedimientos almacenados*, *Disparadores* etc. Incluye la mayor parte de los tipos de datos especificados en los estándares *SQL92* y *SQL99*, como: *entero, numérico, booleano, char, varchar, fecha, interval o timestamp*.

Otras características interesantes de *PostgreSQL* son las siguientes:
-Alta concurrencia que evita tener que bloquear una tabla cuando se está escribiendo en ella.
-Copias de seguridad en línea.
-Replicación asíncrona.
-Transacciones anidadas.
-Optimizador de consultas.
-La característica fundamental del *PostgreSQL* y quizás la más importante es que es libre, no siendo así *MySQL* que es una de las más utilizadas en el mundo pero es software propietario. Esta característica permite confirmar que el gestor a utilizar es *PostgreSQL* y no otro.

Si de cifras se trata, es importante saber que en *PostgreSQL* el tamaño máximo de la base de datos es ilimitado; el de una tabla asciende a 32 TB, el de una fila a 1.6 TB y el de un campo de datos a 1 GB, el número de filas en una tabla es ilimitado, pero no el de columnas, que oscila entre 250 y 1600 columnas por tabla. El número de índices por tabla es también ilimitado.

1.7 El arquitecto de software y los artefactos

El arquitecto del software debe ser polifacético, poseer madurez, visión y una vasta experiencia que le permita tomar decisiones rápidamente y hacer el juicio adecuado y debe ser por ende, crítico en casos de no completitud de la información.

Seguidamente se listan las responsabilidades que establece *RUP* para el arquitecto de software:

- ✓ Priorizar los casos de uso
- ✓ Estructurar modelo de implementación
- ✓ Realizar análisis arquitectónico
- ✓ Construir prueba de concepto de la arquitectura
- ✓ Incorporar elementos de diseño existentes
- ✓ Describir distribución
- ✓ Evaluar viabilidad de prueba de concepto de la arquitectura
- ✓ Identificar mecanismos de diseño
- ✓ Identificar elementos de diseño
- ✓ Describir la arquitectura en tiempo de ejecución

Los artefactos fundamentales a desarrollar según RUP son:

- ✓ Modelo de Análisis
- ✓ Modelo de Diseño
- ✓ Modelo de Despliegue
- ✓ Modelo de Implementación
- ✓ Documento de descripción de la arquitectura

Y en el caso del sistema para la informatización de la facultad sólo se generará el documento de descripción de la arquitectura ya que el mismo contiene la información utilizada para la implementación de dicho sistema.

1.8 Conclusiones

En este capítulo se han abordado temas relacionados con el objeto de investigación, tendencias, metodologías y las tecnologías más utilizadas en el mundo informático y que son examinadas durante el diseño de la arquitectura de software de aplicaciones Web, con el objetivo de proporcionar una solución que permita optimizar la gestión de información y la interoperatividad entre los servicios.

Se han abordado conceptos importantes de la arquitectura de software que son de gran ayuda a la hora de llevar a cabo el desarrollo del sistema. También se ha realizado un estudio de los diferentes estilos arquitectónicos, patrones de software, lenguajes de descripción de la arquitectura y metodologías de desarrollo aplicables a un sistema en conjunto con la selección de los más adecuados para su uso en el desarrollo del sistema. Se llevó a cabo una investigación de los diferentes lenguajes de implementación, tecnologías y herramientas de soporte al desarrollo del sistema, así como los principales artefactos que se generan durante la implementación de un sistema y en los generados para el sistema en cuestión.

CAPÍTULO 2: Descripción de la arquitectura

Introducción

En el presente capítulo se realiza una descripción de la arquitectura propuesta para la integración del sistema a través de un análisis de los requisitos y se hace un análisis de la solución propuesta a través de las *4+1 Vistas Arquitectónicas*. Se comentará acerca de la definición del estándar de codificación a utilizar y los estilos *CSS*. Así como una descripción de las métricas y restricciones arquitectónicas.

2.1 Análisis de los Requerimientos

Requerimientos de Software

Para el desarrollo:

- ✓ Sistema operativo *Windows 2000* o superior / *Linux*.
- ✓ Navegador web *Internet Explorer 6* o superior o *Mozilla Firefox 2.0* o superior.
- ✓ Servidor web *Apache 2.2* o superior, *PHP 5.2* o superior.
- ✓ Servidor de base de batos *PostgreSQL 8.2*.

Para la explotación:

Cliente:

- ✓ Navegador web *Internet Explorer 6* o superior o *Mozilla Firefox 2.0* o superior.

Servidor:

- ✓ Sistema operativo *Windows 2000* o superior/*Linux*.
- ✓ Servidor de base de datos *PostgreSQL 8.2.*
- ✓ Servidor web *Apache 2.2* o superior.

Requerimientos de Hardware

Para el desarrollo:

- ✓ Pentium IV o superior.
- ✓ 256 MB de memoria *RAM* o superior.
- ✓ Disco duro con capacidad de 40 GB o superior.

Para la explotación:

Cliente:

- ✓ Pentium II o superior.
- ✓ 64 MB de memoria *RAM* o superior.
- ✓ Disco duro con capacidad de 20 GB o superior.

Servidor:

- ✓ Pentium IV o superior a 2.0 GHZ.
- ✓ 2 GB de memoria *RAM* o superior.
- ✓ Disco duro con capacidad de 80 GB o superior.

2.2 Vista de Casos de Uso

En esta vista se muestra la percepción que tiene el usuario de las funcionalidades del sistema mediante la representación de los actores y casos de usos más importantes.

Actores:

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

Secretaria Docente:

Usuario con permiso de Administración para el módulo Estudiante, por lo que puede modificar, eliminar e insertar y editar la información de los estudiantes en la aplicación. El sistema debe responder ante cualquier petición.

Jefe Departamento (Jefe de Departamento):

Usuario con permiso para adicionar, eliminar, insertar y modificar datos de los profesores, así como realizar reportes.

Vicedecano Ext_Res (Vicedecano de Extensión Universitaria y Residencia):

Usuario con permiso de crear actividades, editarlas y eliminarlas, de ubicar estudiantes en la residencia y gestionar la información de los edificios y apartamentos.

Vicedecano Prod (Vicedecano de Producción):

Usuario con permiso para gestionar polos, proyecto, maquinas y la asignación de tesis, tutores y oponentes; tiene acceso total en el módulo Producción.

Casos de Usos Arquitectónicamente Significativos:

Los casos de uso describen un conjunto de secuencias de acciones, incluyendo variaciones que un sistema lleva a cabo y que conduce a un resultado observable de interés para un determinado actor. En el caso de los arquitectónicamente significativos, son aquellos que representan partes críticas del sistema. Los referentes al sistema *SIGIPE* han sido priorizados debido al soporte que los mismos brindan al negocio.

Diagrama de Casos de Uso Arquitectónicamente Significativos:

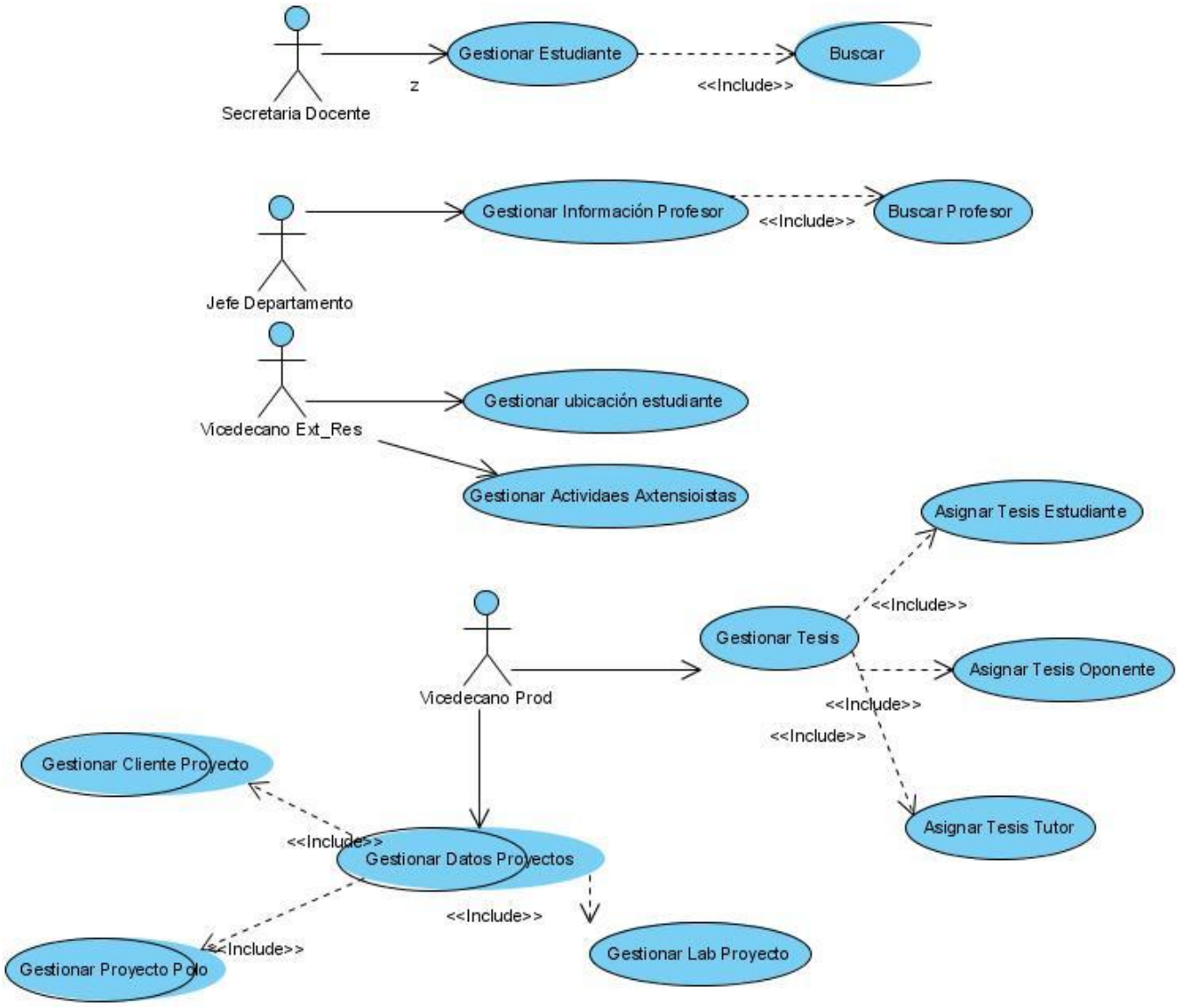


Fig.1 Diagrama de Casos de Uso Arquitectónicamente Significativos.

Este diagrama muestra la interacción de los actores de la aplicación con los principales casos de uso del sistema, es decir, aquellos que son significativos. Mostrando además las relaciones que se establecen entre los casos de uso, ya sean las relaciones de inclusión o de extensión.

Breve descripción de los Casos de Uso Arquitectónicamente Significativos:

CU: Gestionar Actividades de extensión

Descripción:

El caso de uso tiene lugar cuando el vicedecano de extensión accede al sistema para realizar cualquier operación que éste soporta. Para lograr esto debe autenticarse correctamente y luego podrá gestionar las actividades, es decir, crearlas y modificarlas o eliminarlas.

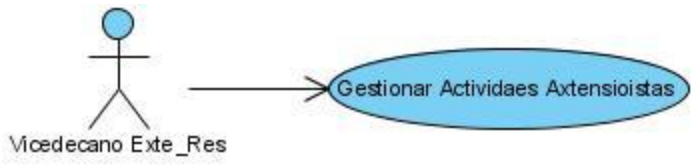


Fig.2 Diagrama de Caso de Uso Arquitectónicamente Significativo: CU Gestionar Actividades de extensión. Módulo Extensión.

CU: Gestionar_Ubicacion_Estudiante

Descripción:

El caso de uso tiene lugar cuando el vicedecano de extensión accede al sistema para realizar cualquier operación que éste soporta. Para esto debe haberse autenticado correctamente, luego selecciona la opción de insertar, eliminar, modificar o mostrar estudiante, así como la de darle una ubicación al estudiante.

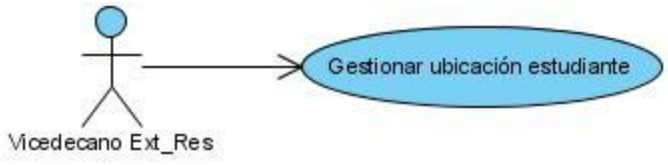


Fig.3 Diagrama de Caso de Uso Arquitectónicamente Significativo: CU Gestionar Ubicación Estudiante. Módulo Residencia.

CU: Gestionar_Informacion_Profesores

Descripción:

El caso de uso tiene lugar cuando el jefe de departamento accede al sistema para realizar cualquier operación que éste soporta. Para esto debe haberse autenticado correctamente con anterioridad y luego selecciona la opción de modificar, insertar, eliminar o mostrar los datos de profesores.

Fig.4 Diagrama de Caso de Uso Arquitectónicamente Significativo: CU Gestionar Información Profesor. Módulo Profesor.

CU: Gestionar_Estudiante

Descripción:

El caso de uso tiene lugar cuando la secretaria docente accede al sistema para realizar cualquier operación que éste soporta. Para esto debe haberse autenticado correctamente previamente a la gestión de la información de los estudiantes, dígase insertar, eliminar, modificar o mostrar datos de los estudiantes.

Fig.5 Diagrama de Caso de Uso Arquitectónicamente Significativo: CU Gestionar Estudiante. Módulo Estudiante.

CU: Gestionar_Datos_Proyecto

Descripción:

El caso de uso tiene lugar cuando el vicedecano de producción accede al sistema para realizar cualquier operación que éste soporta. Para esto debe haberse autenticado correctamente con anterioridad para luego manipular los datos de los proyectos. Tiene la opción de insertar, eliminar, modificar o mostrar laboratorios, proyectos, máquinas y polos. Tiene acceso total en este módulo.

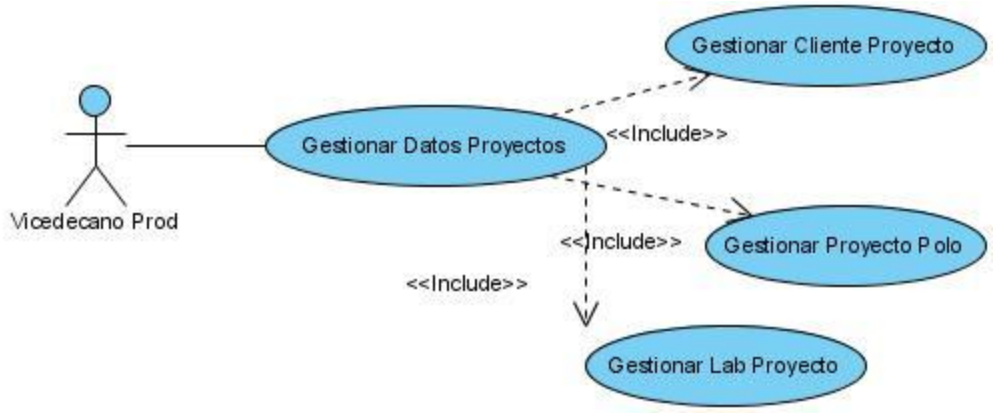


Fig.6 Diagrama de Caso de Uso Arquitectónicamente Significativo: CU Gestionar Datos Proyecto. Módulo Producción.

CU: Gestionar_Tesis

Descripción:

El caso de uso tiene lugar cuando el vicedecano de producción accede al sistema para realizar cualquier operación que éste soporta. Para esto debe haberse autenticado correctamente, luego puede seleccionar la opción de insertar, modificar, asignar tesis a oponente, a estudiantes y a tutores con el fin de gestionar esta información. Tiene acceso total en este módulo.

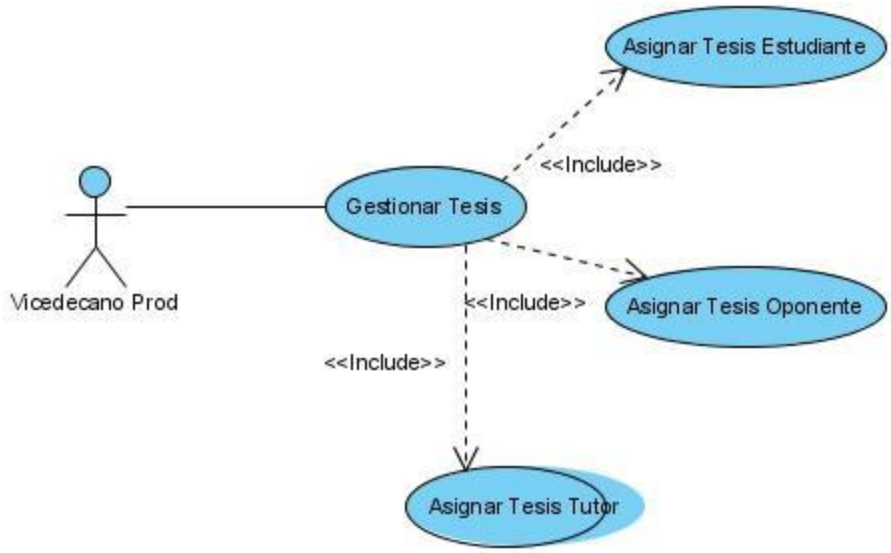


Fig.7 Diagrama de Caso de Uso Arquitectónicamente Significativo: CU Gestionar Tesis. Módulo Tesis.

2.3 Vista lógica

La presente vista de arquitectura del modelo de diseño o *Vista Lógica* como también se le conoce, representa los subsistemas e interfaces más importantes para la arquitectura del sistema en cuestión.

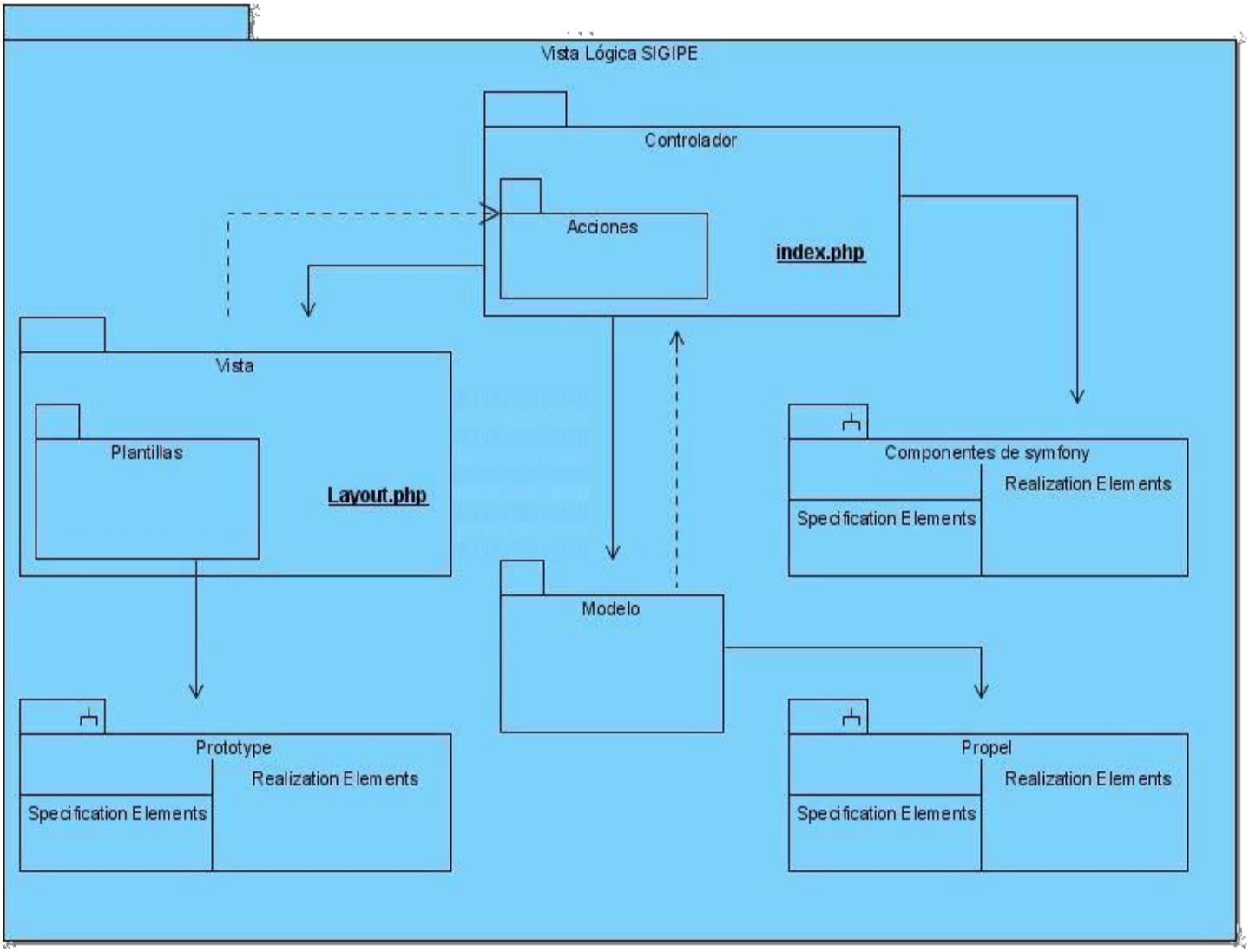


Fig.8 Estructura de la Arquitectura. Vista Lógica de la aplicación *SIGIPE*.

El sistema *SIGIPE* se encuentra organizado mediante la utilización del patrón arquitectónico *Modelo Vista Controlador* implementado por el framework *Symfony* propuesto para el desarrollo del sistema. En esta se puede apreciar la relación entre las diferentes capas que componen la vista, es decir, la interacción entre el controlador, la vista y el modelo, así como también la relación con las diferentes librerías a utilizar para facilitar el trabajo con el framework.

Capa de Presentación o Vista:

La *Vista* es la capa que interactúa con el usuario, es la que se encarga de mostrar y manejar cualquier evento del mismo, para posteriormente enviar la petición al su capa inferir (*Controlador*) y mostrar una respuesta en dependencia de su petición. Es en la capa de *Presentación* o *Vista* donde se realizan las primeras validaciones de las entradas del usuario. En este caso, la capa *Vista* esta interactuado con un subsistema llamado *Prototype*, que contiene funcionalidades, tanto para validaciones como para los estilos *CSS*.

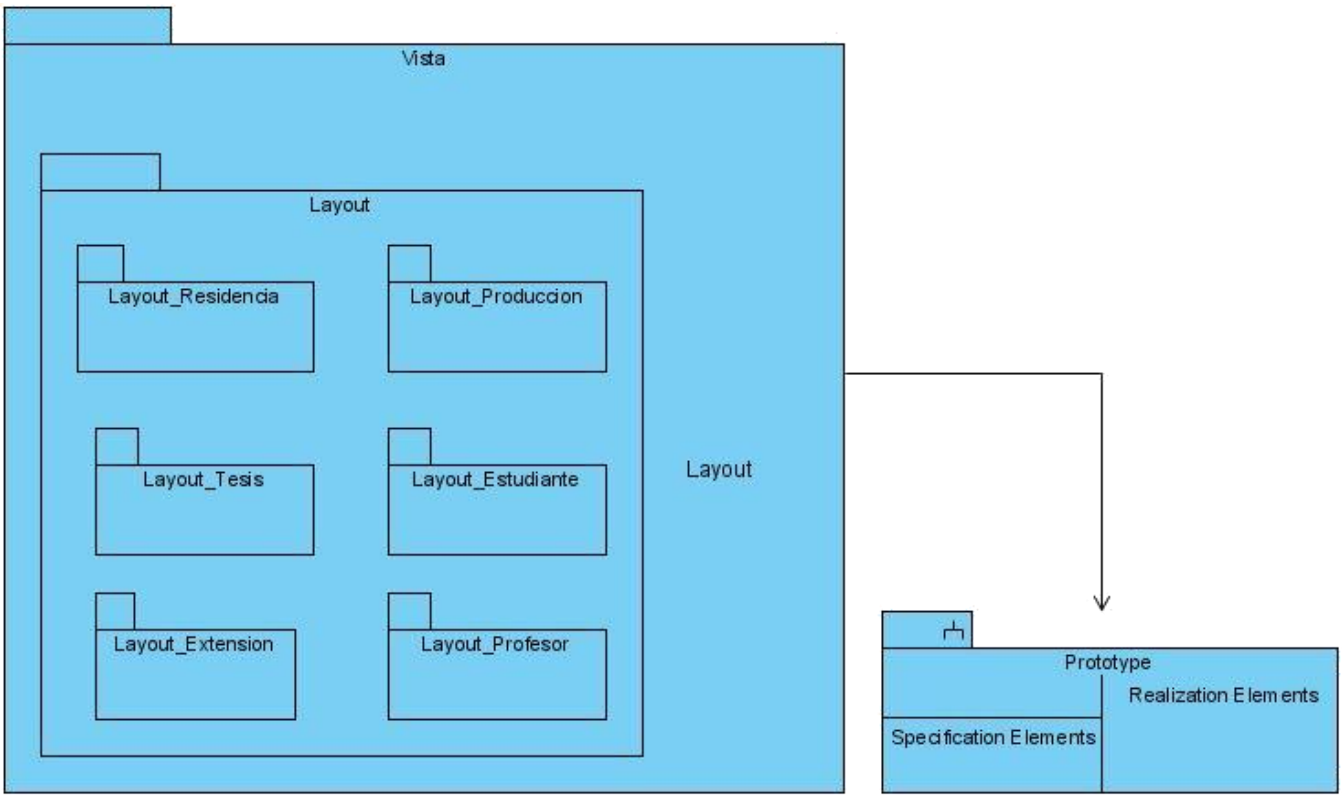


Fig.9 Capa Vista.

Capa Controlador:

La capa de *Negocio* o *Controladora* contiene las funcionalidades del negocio que dan respuesta a los requerimientos. Recibe peticiones de la capa de Presentación o *Vista* y procesa la lógica de negocio basada en las peticiones de los usuarios. Es la capa que media entre la *Vista* y el *Modelo*. Es donde se encuentran todas las acciones que se realizaran y que tiene implementada la aplicación. En esta capa podemos ver el paquete de acciones de cada módulo y cómo interactúan con el subsistema de *Symfony* que facilita que se cumplan muchas de estas acciones.

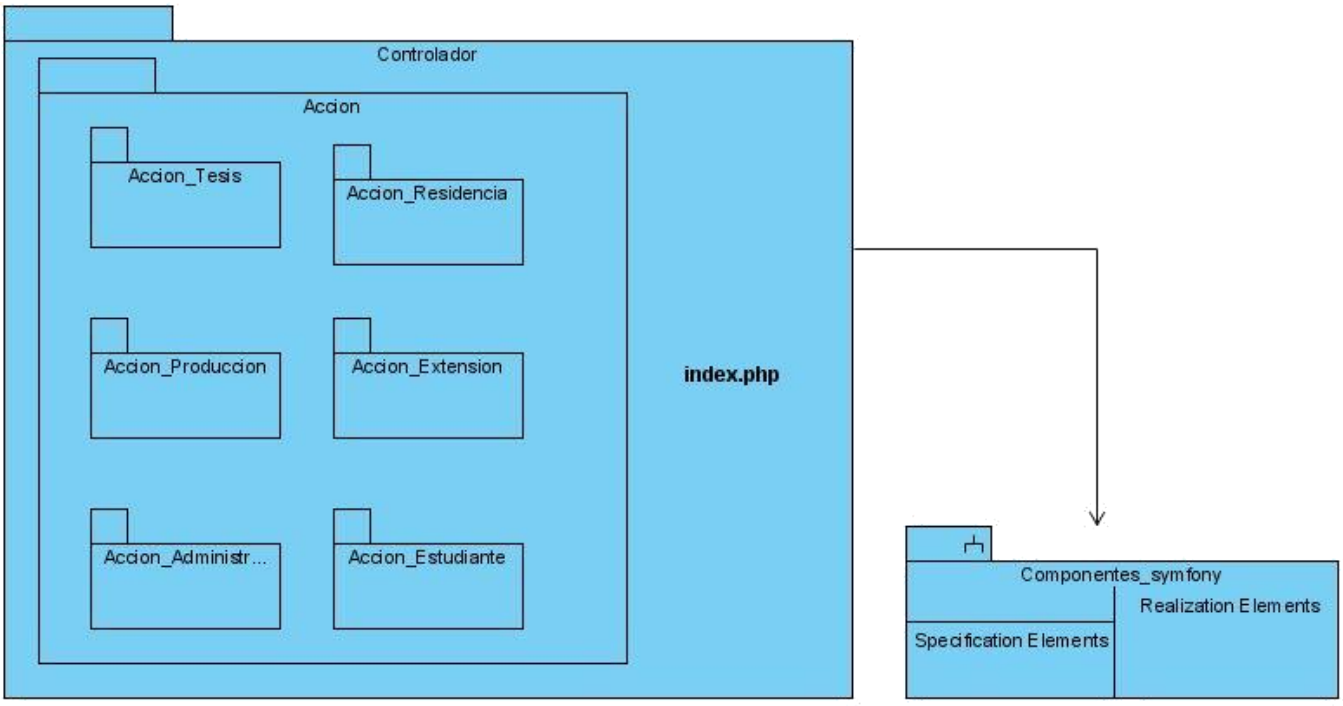


Fig.10 Capa Controladora.

Capa Modelo:

La capa *Modelo* es la que contiene las clases persistentes, de las que cada una tiene correspondencia con una tabla de la base de datos. La capa *Modelo* es la contenedora de información, la cual es manejada por la capa *Controladora* para mostrarla en la capa de *Presentación* o *Vista*. Esta capa muestra las clases entidades o persistentes y por cada una de estas clases varias clases más que se encuentran dentro del subsistema *Propel* que son las que se encargan de la conexión y consultas con la base de datos.

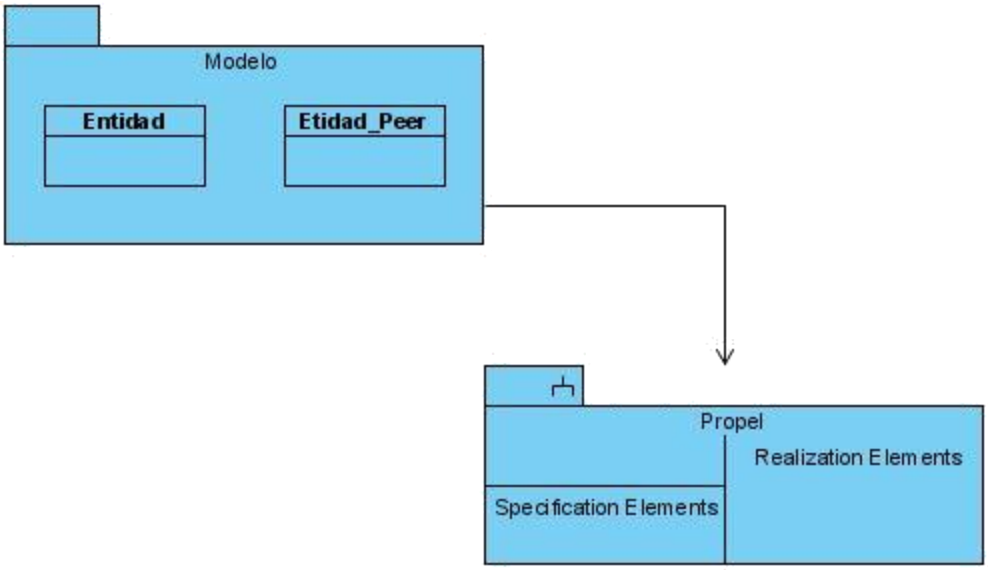


Fig.11 Capa Modelo.

2.4 Vista de Implementación

En la siguiente vista se muestra y describe el conjunto de subsistemas, paquetes y componentes de implementación en que se desdobla la aplicación *SIGIPE*. Esta vista puede realizarse con la ayuda del *Modelo de Implementación*, artefacto principal del flujo de implementación definido por la metodología *RUP*. En esta vista se representan también las relaciones entre las diferentes capas en que está implementada la lógica del sistema.

Vista de implementación de la aplicación *SIGIPE*.

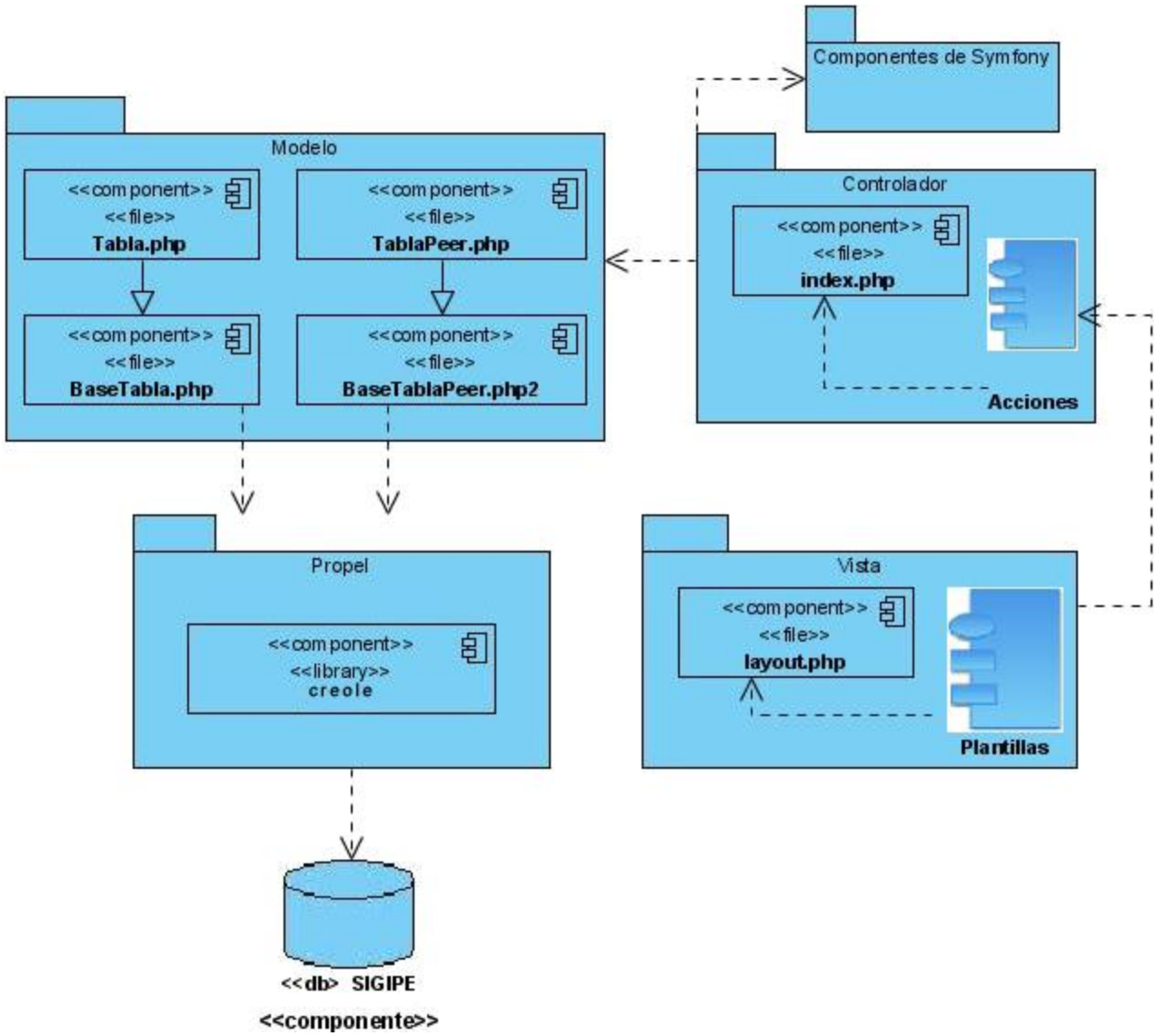


Fig.12 Vista del árbol de directorio que propone Symfony y por tanto adoptado por el sistema.

A continuación se procede con una descripción de cómo se refleja la implementación en cada uno de los subsistemas de implementación:

Subsistema *Vista*:

Este subsistema contiene el diseño general *(Layout)* de la aplicación, las diferentes plantillas o interfaces con las que interactúan los usuarios directamente y la relación entre ambos, siendo el *Layout* un elemento común donde se muestran las plantillas. Estas plantillas constituyen a su vez el medio de comunicación entre los clientes y el *Controlador*.

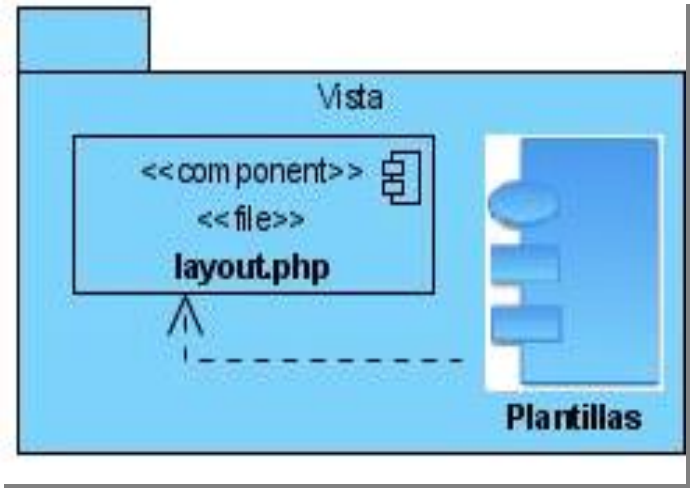


Fig. 13 Subsistema de implementación Vista.

Subsistema *Controlador*:

El *Controlador* es el subsistema que contiene todas las funcionalidades a las cuales pueden acceder los usuarios en dependencia de las restricciones que tengan las mismas para ciertos usuarios teniendo en cuenta sus niveles de acceso a las mismas. Este subsistema hace uso de los componentes necesarios para el correcto funcionamiento de las funciones implementadas en el mismo. Este subsistema es por donde pasan todas las peticiones de los clientes antes de acceder al modelo.

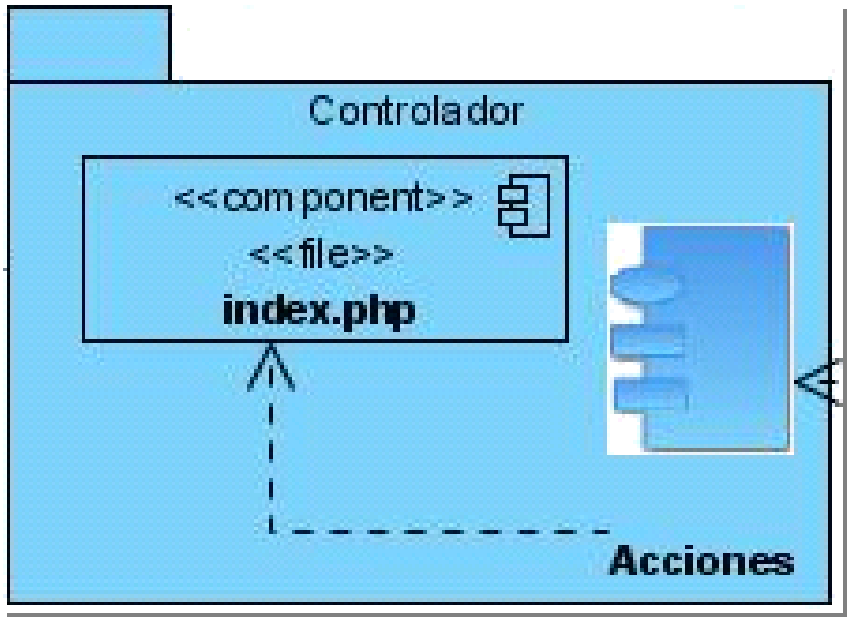


Fig. 14 Subsistema de implementación Controlador.

Subsistema *Modelo*:

En este subsistema se encuentran diferentes componentes que representan las tablas de la base de datos mapeadas a clases mediante el uso de *Propel*. Estas clases son las que utiliza el framework para acceder a los datos a través de las librerías *Creole* usada por *Propel*. En este subsistema se puede apreciar la relación de herencia que se establece entre las clases las clases nombradas como las tablas y sus clases bases, así como las clases *Peer* de las tablas y sus clases *Bases-Peer* correspondientes, originadas todas a partir de las tablas de la base de datos.

Las clases bases contienen el mapeo de los campos de las tablas a atributos de las mismas, quedando en las clases *Bases-Peer* las relaciones entre los atributos y los campos de las tablas.

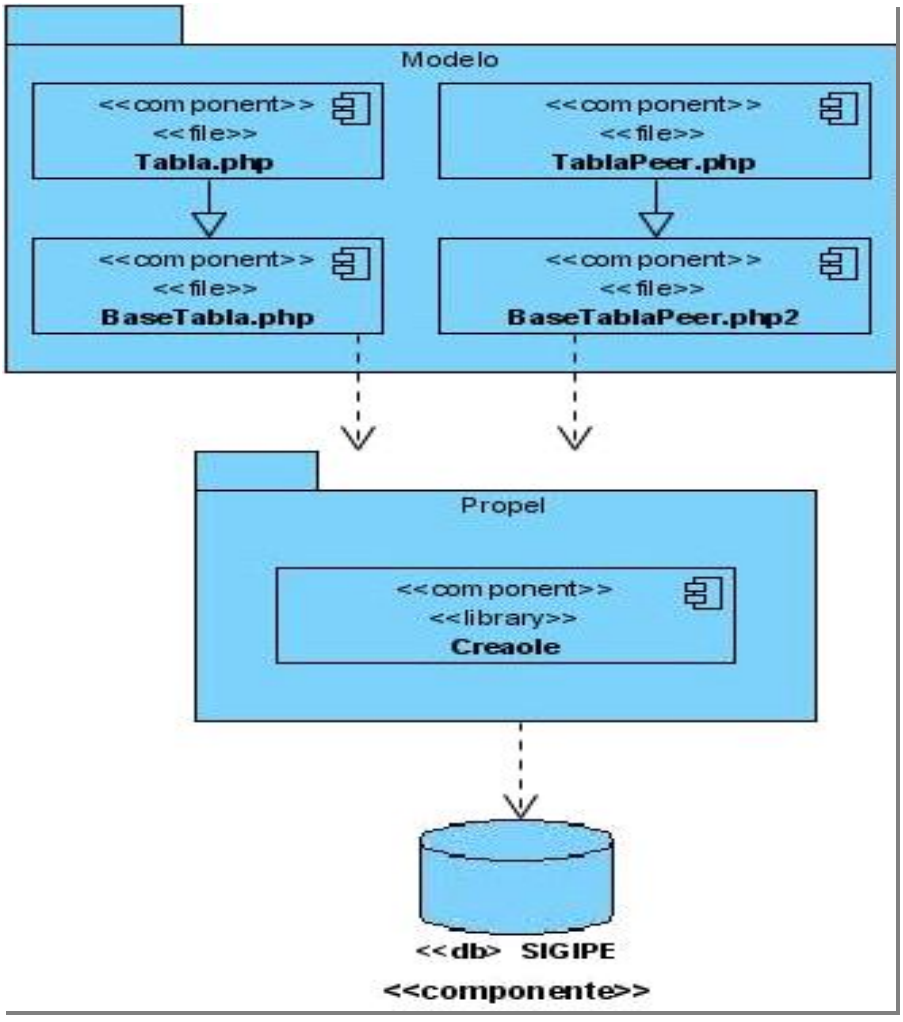


Fig. 15 Subsistema de implementación Modelo.

Vista del árbol de directorio de la aplicación *SIGIPE*:

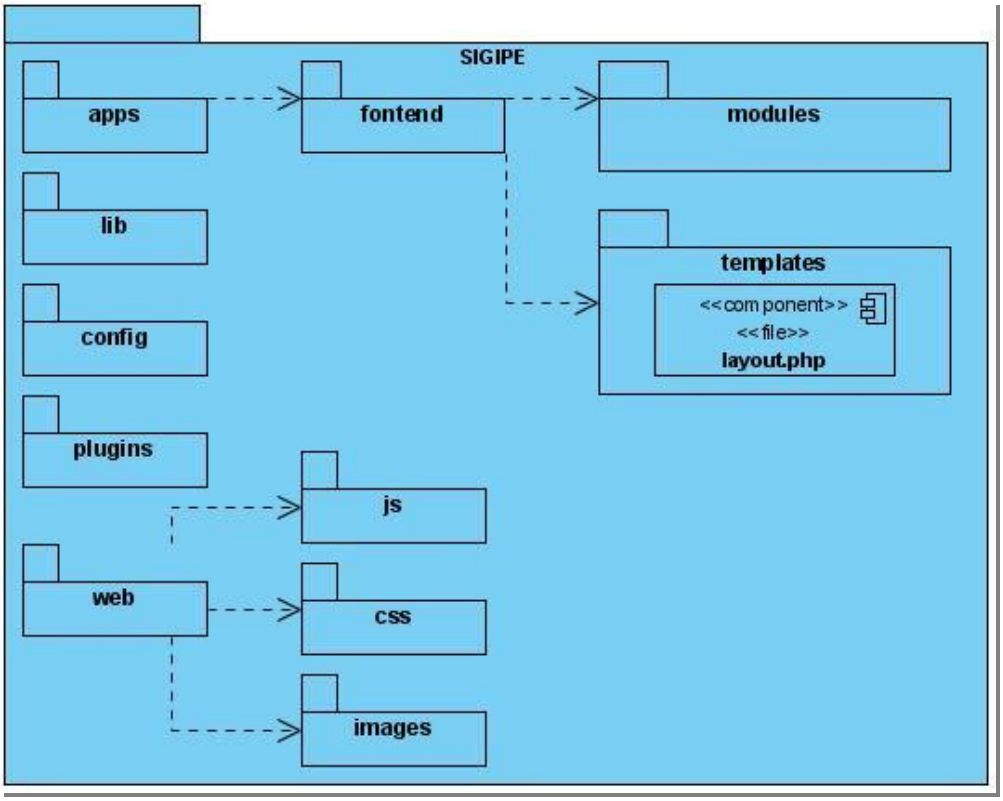


Fig.16 Vista del árbol de directorio de la aplicación SIGIPE.

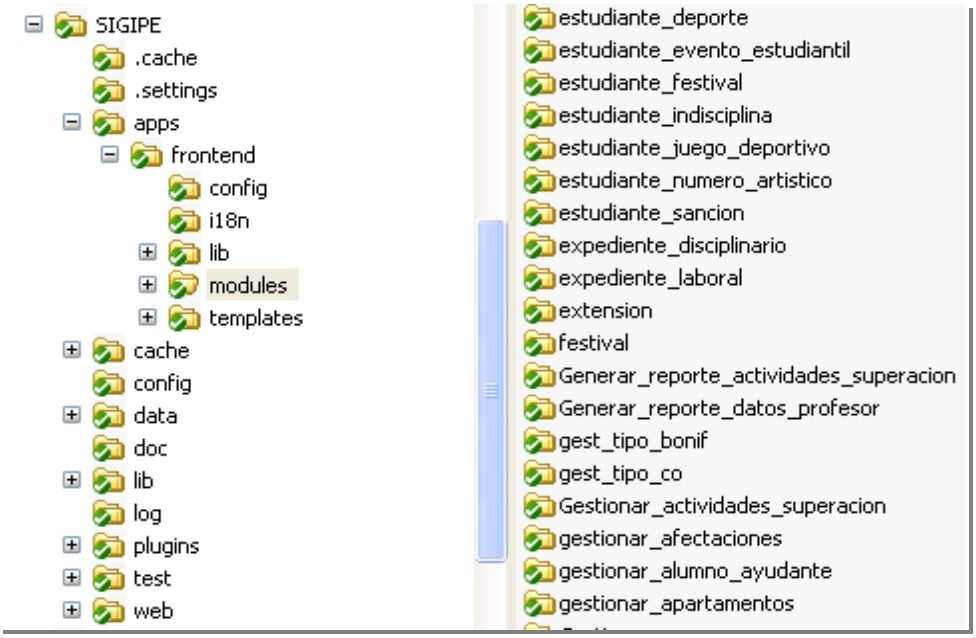


Fig. 17 Vista del árbol de directorio de la aplicación SIGIPE desde el explorador de Windows. (Para ver esta vista con la totalidad de los módulos ver anexo 3)

En esta vista se refleja la estructura del proyecto en carpetas, siendo la carpeta *SIGIPE* la carpeta raíz del proyecto. Esta carpeta raíz contiene en un segundo nivel las subcarpetas que a continuación se describen:

web: En esta subcarpeta están contenidas la subcarpeta *css* para mantener en ella los archivos con los estilos a utilizar en la aplicación, la subcarpeta *js* para los archivos con las funciones que se ejecutarán en las páginas clientes y la subcarpeta *images*, en la cual se almacenarán las imágenes a utilizar en el diseño de la aplicación. Esta subcarpeta también puede contener otras subcarpetas con fines web que necesiten los desarrolladores.

lib: En esta subcarpeta se encuentran las clases del modelo generadas a partir de las tablas de la base de datos, así como los formularios generados para la entrada de los datos.

config: Esta subcarpeta es la contenedora de los archivos de configuración del proyecto. Aquí se puede encontrar el *databases.yml* para la especificación de la base de datos a utilizar así como el host donde se encuentra la misma. Se encuentra también el schema.yml el cual es el modelo de objeto que utiliza el *ORM (Object-Relational Mapping)* para llevar a cabo el mapeo y contiene la definición de las tablas, sus relaciones y las características de sus columnas.

plugins: Esta subcarpeta agrupa los diferentes *plugins* que se instalan para su uso en el proyecto. En el caso del sistema en cuestión, el *plugin* que se encuentra en este directorio es el *sfGuardPlugin* utilizado para implementar la seguridad del sistema.

apps: Esta subcarpeta contiene la aplicación frontal del sistema, la cual contiene a su vez la subcarpeta modules que contiene los 107 sub-módulos, la subcarpeta templates que contiene las plantillas que se van a cargar en el layout y una carpeta para configuraciones que permiten definir el layout que se va a utilizar.

Dentro de la subcarpeta modules hay 107 sub-módulos como resultado de que el sistema se integra a nivel de módulos y no de aplicación. Esto quiere decir que se crea una sola aplicación que contiene todos los módulos principales y los sub-módulos en vez de una aplicación por cada módulo principal que contenga sus respectivos sub-módulos. Esta estrategia se llevó a cabo ya que la información que se maneja de los estudiantes y profesores de la facultad 6 es muy voluminosa y de esta manera se

gana en agilidad a la hora de dar una respuesta por parte del sistema, además de que a nivel de módulos se pueden compartir mejor los componentes y garantizar mayor reutilización de código que a nivel de aplicación.

Es preciso notificar que la integración a del sistema desde la aplicación se verá reflejada en el menú de la misma que garantiza acceso a las funcionalidades de los módulos. Este menú se encuentra en un layout que resulta común para todas las plantillas de los módulos. En caso de tener más de una aplicación resultaría trabajoso mostrar en un solo layout las plantillas de todas las aplicaciones, además de que habría que implementar la seguridad en cada una de estas aplicaciones por separado, no siendo así en la estrategia de integrar todos los módulos en una misma aplicación.

El principal inconveniente que esta estrategia trae consigo es la dificultad a la hora de buscar un determinado módulo ya que se hace engorroso encontrarlo debido a la gran cantidad de módulos en la misma subcarpeta. Para solucionar esta situación se creó una interfaz web que muestra los vínculos de acceso a los módulos principales y a sus respectivos sub-módulos. Es preciso aclarar que esta interfaz no será accesible desde ninguna de las interfaces de la aplicación SIGIPE, ya que ésta solo representa una herramienta para facilitar el acceso de los desarrolladores a los diferentes módulos y sub-módulos de la aplicación durante el desarrollo de la misma y también en el momento que sea necesario buscar algún módulo para su mantenimiento luego del despliegue.

A continuación se muestran una serie de imágenes de esta interfaz.

Fig. 18 Interfaz web con vínculos a los módulos.

Fig. 19 Interfaz web que muestra los sub-módulos del módulo Profesor.

2.5 Vista de Despliegue

La vista de despliegue permite establecer la distribución física del sistema en nodos de procesamiento, así como los componentes que lo integran y las formas de conexión que se establecen entre ellos.

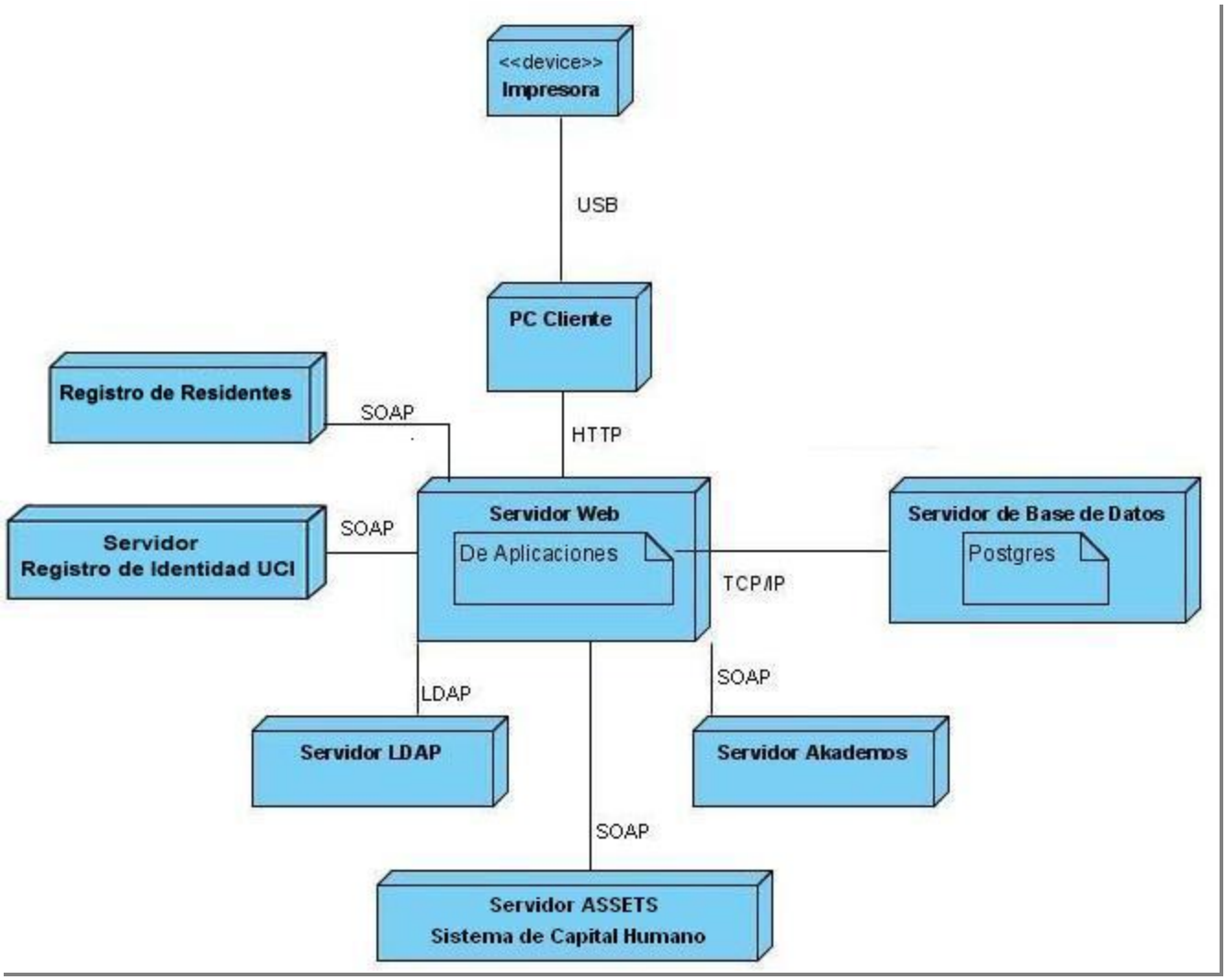


Fig.20 Diagrama de Despliegue

A continuación se plasma una descripción de los nodos que componen el despliegue del sistema.

PC Cliente: Este nodo representa las computadoras que usarán los clientes para acceder al sistema a través de la interfaz web diseñada y mediante el protocolo *HTTP*.

Impresora: Este nodo representa el dispositivo a utilizar para la impresión de reportes en el local que se disponga. La misma estará conectada mediante el puerto *USB* a la computadora del cliente con la responsabilidad de realizar las impresiones.

Servidor web: Este nodo es el destinado a alojar la aplicación. Al mismo se conectan las computadoras de los clientes y éste a su vez se conecta a los distintos servidores que ofrecen servicios web, de los cuales se consume información para realizar las actualizaciones y sincronizaciones de la base de datos de sistema *SIGIPE*.

Servidor de bases de datos: En este nodo se encontrará alojada la base de datos del sistema, la cual estará montada sobre el gestor *PostgreSQL*.

Para el despliegue del sistema se concibió mantener el servidor de la aplicación y el servidor de la base de datos en diferentes servidores por petición de la asesoría de arquitectura de la facultad, ya que de este modo también se garantiza compartir la carga de trabajo entre ambos servidores para obtener un mejor rendimiento.

Servidor Akademos: Este nodo representa el servidor del cual se consume el servicio que ofrece los datos de los estudiantes de la facultad.

Servidor LDAP: Este nodo representa el servidor que ofrece el servicio para la autenticación por el dominio *UCI*.

Registro de Residentes: Este nodo representa el servidor del cual se consume el servicio para la obtención de información tanto de los profesores como de los estudiantes.

Registro de Identidad *UCI*: Este nodo representa al servidor que facilita la obtención de los usuarios del dominio *UCI*.

Sistema de Capital Humano: Este nodo representa el servidor del cual se obtiene información de los profesores de la facultas 6.

2.6 Modelo de Datos

El modelo de datos del sistema aporta la base conceptual para llevar a cabo el diseño de la aplicación, así como la base formal para las herramientas y técnicas empleadas en el desarrollo y uso del sistema de información *SIGIPE*. Además nos ayuda a expresar las propiedades estáticas y dinámicas de la aplicación para el posterior uso intensivo de los datos.

El presente modelo de datos de la base de datos *SIGIPE* presenta la integración de las seis bases de datos correspondientes a los módulos *(Estudiante, Profesor, Tesis, Residencia, Extensión y Producción)* del sistema. Para la integración de las bases de datos se tuvo en cuenta la selección de las tablas principales *(Estudiante y Profesor)* alrededor de las cuales se centra toda la información a almacenar en la base de datos ya integrada y que eran comunes a los seis módulos, respetando además para la integración las relaciones ya previstas entre las tablas de cada módulo individualmente. De esta manera se garantiza la correspondencia en las relaciones una vez llevada a cabo la integración de las bases de datos.

Una de las tareas acordadas en la investigación dentro del grupo de tareas para dar respuesta al problema científico planteado es la definición de la actualización y sincronización de la base de datos de la aplicación con las diferentes bases de datos de las cuales inicialmente se hace uso para poblar la misma y luego mantenerla actualizada.

Las bases de datos usadas para sustentar la base de datos *SIGIPE* son actualizadas con cierta frecuencia para garantizar la confiabilidad de los datos. En el caso de las actualizaciones y sincronizaciones de la base de datos *SIGIPE* se llevaran a cabo cada quincena del curso, es decir, dos veces al menos cada mes y a nivel de aplicación, es decir, las actualizaciones se llevarán a cabo desde la aplicación y por la persona con los permisos o credenciales pertinentes para esta acción. Estas actualizaciones garantizarán que la información con la que se trabaja en la facultad esté actualizada y sea confiable. Puede que este período de tiempo entre una actualización y otra parezca muy corto en cierto sentido, pero el motivo por el cual se propone este margen de tiempo es porque en ocasiones los estudiantes y/o profesores causan baja de la facultad, ya sea por traslado hacia otra facultad o por baja de la universidad, y de no tener la base de datos actualizada se estaría trabajando con información que no es real y por tanto poco confiable.

La información que se va a actualizar es la relacionada con los estudiantes y profesores, ya que alrededor de estos se centra la información que se manipula en la facultad. En el caso de las actualizaciones de los datos de los estudiantes podrá ser llevada a cabo desde el módulo Estudiantes por años académicos o por la matrícula total de la facultad en dependencia de cómo resulte más cómodo realizar la actualización o de las prioridades de la misma, siendo la secretaria docente la encargada de esta responsabilidad. En el caso de la actualización de la información profesoral será llevada a cabo desde el módulo Profesores según la matrícula de profesores de la facultad y al igual que en el caso anterior por la persona encargada de esta tarea, es decir, alguno de los jefes de departamento.

Es preciso tener en cuenta que estas actualizaciones pueden hacerse diariamente si se desea pero esto es poco práctico y carece de sentido, pues no todos los días los estudiantes y profesores causan baja de la facultad por traslado o baja del centro.

Dado al gran número de tablas con que cuenta la base de datos, un total de 127, a continuación solo se relaciona una muestra del modelo de datos de la base de datos, en el cual solo están contenidas algunas de las principales tablas de la misma como son la tabla *Estudiante, Profesor, Edificio, Apartamento, Tesis, Proyecto* y la tabla *Cliente* entre otras. Al igual que con las tablas, solo se muestran algunas de las relaciones entre las mismas.

Para una total completitud de la información relacionada con el modelo de datos de la base de datos del sistema *SIGIPE*, dirigirse a la sección de los anexos y consultar el anexo 2.

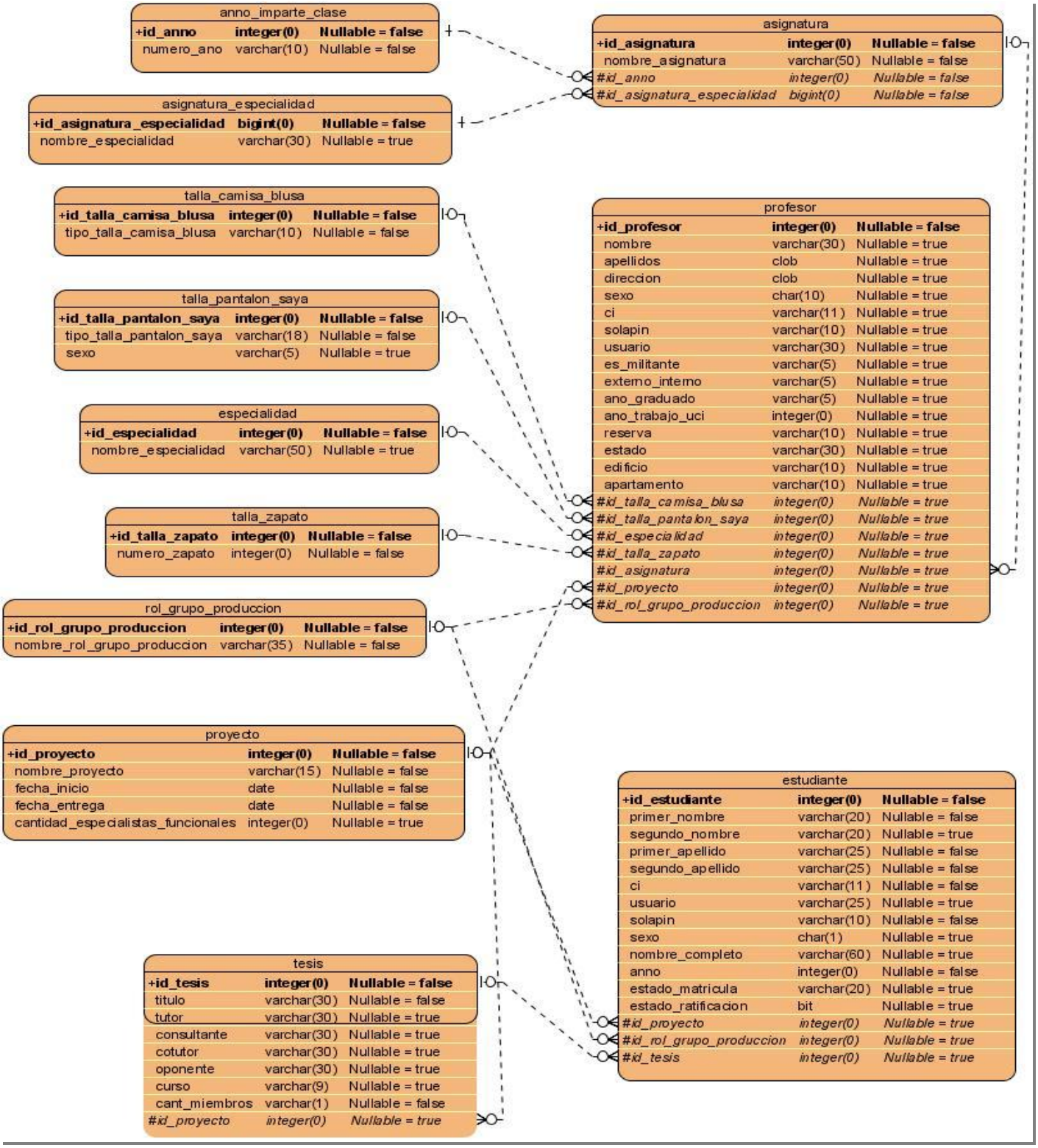


Fig. 21 Muestra de la vista del modelo de datos de la base de datos SIGIPE.

2.7 Metas y restricciones arquitectónicas

Apariencia o interfaz externa:

Para la interfaz se utilizarán colores frescos en la tonalidad verde que no carguen el diseño y con buen contraste entre los mismos.

Usabilidad.

La aplicación podrá ser utilizada por personal vinculado al manejo de información de Profesores y Estudiantes de la facultad 6 de la *UCI*.

Soporte y portabilidad.

El sistema podrá ser usado sobre los sistemas operativos Windows y Linux.

Seguridad.

El sistema debe contar con protección contra acciones no autorizadas o que puedan afectar la integridad de los datos que en él se manejan.

Políticos-culturales.

La aplicación se desarrollará haciendo uso del idioma español y será desplegada bajo el uso de este idioma.

Legales.

Para desarrollo de la aplicación se hará uso de herramientas de software libre.

Confiabilidad.

El sistema contará con la seguridad requerida para mantener la integridad de los datos y estará disponible para su acceso en cualquier momento que se requiera manteniendo así su disponibilidad, siendo así confiable para su uso.

2.8 Estándar de codificación a utilizar en la implementación del sistema.

Variables:

- ✓ Las variables globales se iniciarán con la letra “g” seguidas del carácter guión bajo “_”.

ej. g_variable.

- ✓ Siempre se iniciarán con minúscula.
- ✓ Si el nombre de la variable es compuesto se unen por el carácter guión bajo “_”.
- ✓ Las variables para arreglos se inician con la palabra “arreglo” y el guión bajo.

ej. arreglo_estudiantes.

- ✓ Las variables locales se declararán en su forma sencilla, con un nombre descriptivo y corto.

ej. **var** edad.

- ✓ Cuando se declara una variable al lado se comenta en una línea su uso a cuatro (4) espacios de distancia.

ej. **var** cantidad_estudiantes; //almacena el valor de la cantidad de estudiantes

- ✓ Las variables se declararán en líneas independientes siendo permisible en los casos que lo requiera hasta tres (3) variables del mismo tipo en una línea.

ej. **var** edad, anno, facultad;

Funciones:

- ✓ Las funciones se iniciarán con la palabra “execute” seguido del nombre de la función iniciado con letra mayúscula.

ej. executeFuncion().

- ✓ Los nombres de las funciones serán cortos y descriptivos. Si alguno es compuesto se separará por el guión bajo y cada palabra inicia con mayúscula siguiendo el estilo de encamellado.

ej. executeBuscar_Estudiante().

- ✓ Se insertará un comentario de la función a cuatro (4) espacios de distancia de su declaración.

ej. **function** executeBuscar_Estudiante (id_estudiante) //Función para buscar un estudiante teniendo su identificador.

- ✓ Las llaves de apertura de las funciones estarán justo debajo de la primera letra con que inicia la declaración de la función y en la línea siguiente de la misma.

- ✓ El cuerpo de las implementaciones estarán a la distancia de una tabulación de cuatro (4) espacios.
- ✓ Las llaves de cierre de las funciones estarán en línea vertical con las de apertura y en la línea siguiente de la terminación de la función.
- ✓ Las funciones estarán a la distancia de dos (1) líneas entre las mismas.

Ej.

```
function executeBuscar_Estudiante(id_estudiante)        //Función para buscar un estudiante teniendo su identificador.
{
    var posición;    //controla la posición de un estudiante
}

public function executeInsertar_Estudiante(e)    //Función para insertar un estudiante teniendo sus datos.
{
   var variable;
}
```

- ✓ En el caso de las funciones donde se realizan operaciones matemáticas se incluirán espacios a ambos lados de los operadores usados.

ej. y = 15 * x + promedio;

2.9 Definición de los estilos CSS a utilizar en el diseño de la aplicación del sistema.

Estilos para las tablas que muestran los valores:

- ✓ Usarán *cell-spacing* de al menos *2px* y hasta *10px* en los casos que el volumen de información a mostrar lo requiera por quedar muy unida.
- ✓ Las tablas llevaran el estilo “cuerpo_tabla”.
- ✓ El encabezado de las tablas (primera fila con el nombre de la tabla) tendrá el estilo “cabecera_tabla”.
- ✓ El nombre de las etiquetas de los datos tendrá el estilo “texto”.
- ✓ Los valores a mostrar tendrán el estilo “css_valores”.

Estilos para los formularios de entrada de datos:

- ✓ Dentro tendrán una tabla con tres columnas con el estilo para las tablas; la primera columna para el nombre del dato a recoger y alineado a la derecha. La segunda columna con el componente apropiado para el dato a recoger y alineado a la izquierda. La tercera columna para posibles mensajes de error.
- ✓ Las etiquetas de cada campo (p.e: **Nombre:**) tendrán los estilos “texto”.
- ✓ Los campos de texto “css_campo_texto” y alineados a la izquierda.
- ✓ Los campos de error dinámicamente se le asignara una clase css llamada “css_campo_error”, para esto se adiciona un “td” en el formulario donde puede ocurrir un error y se le pone la clase antes mencionada.
- ✓ Los Checkbox, Select y los Radio no llevan clase pues usarán el estilo definido para el componente correspondiente.
- ✓ En el caso de los Select, el primer elemento o valor siempre será <<Selecciones>>.

Estilos para los botones:

Todos los botones serán de tipo Button, ninguno en los formularios de entrada de datos tendrá imágenes a no ser los que están en las páginas que muestran datos, que se les pondrá una imagen que indica adicionar, editar, ver, eliminar y volver.

- ✓ En el caso de los que adicionan llevarán esta imagen: /images/add.png
- ✓ En el caso de los que editan llevarán esta imagen: /images/editar.png
- ✓ En el caso de los que muestran llevarán esta imagen: /images/ver.png
- ✓ En el caso de los que eliminan llevarán esta imagen: /images/eliminar.png
- ✓ En el caso de los que indican volver llevarán esta imagen: /images/atras.png

Los botones se nombrarán así de la siguiente manera:

- ✓ “Aceptar” los que aceptan una acción dentro de un formulario y llevarán la clase “boton_acep”.
- ✓ “Editar” los que editan información contenida en los formularios.
- ✓ “Cancelar” los que cancelan una acción dentro de un formulario y usaran la clase “boton_canc”.
- ✓ “Volver” los que indican volver a otra página.
- ✓ Los botones estarán siempre alineados al centro y separados por dos espacios en caso de ser más de uno.

Para la paginación se usaran las imágenes que se encuentran en los siguientes directorios:

- ✓ Para el botón “Siguiente”: /images/paginacion/next.png
- ✓ Para el botón “Último”: /images/paginacion/last.png
- ✓ Para el botón “Anterior”: /images/paginacion/previous.png
- ✓ Para el botón “Primero”: /images/paginacion/first.png
- ✓ Los vínculos de los resultados llevaran la clase: “paginacion”.

Clases para aplicar los estilos para los mensajes:

Los mensajes de notificación, que no son los de error, llevarán la clase “notificacion”.
Los mensajes que notifican una acción exitosa usarán la clase “accion_exitosa”.

(Algunas palabras no se tildaron de manera intencional ya que las mismas son el propio nombre de la clase).

2.10 Conclusiones

En este capítulo se realizó la descripción de la arquitectura propuesta para el sistema, haciendo referencia a los requerimientos no funcionales de hardware y software requeridos tanto para el desarrollo como para la puesta en marcha del sistema, así como las diferentes vistas que propone *RUP*; la vista de *CUS*, donde interactúan los actores con los casos de uso más significativos, la vista lógica, donde el sistema se divide en componentes y subcomponentes y como se aplica el patrón *Modelo Vista Controlador*, la vista de despliegue donde se muestra como estará desplegada la aplicación en nodos de procesamiento. También se hace mención a las metas y restricciones arquitectónicas a través de requisitos no funcionales como usabilidad, apariencia e interfaz externa, usabilidad, confiabilidad entre otros.

CAPÍTULO 3: Evaluación de la Arquitectura.

Introducción

En todo desarrollo de software es necesario conocer si éste cumple o no con al menos uno de los atributos de calidad especificados en los requerimientos no funcionales. Para evitar riesgos innecesarios es recomendable realizar evaluaciones a la arquitectura durante su diseño.

La realización de un buen diseño de arquitectura de software garantiza que el sistema cumpla con uno o varios atributos de calidad, y dicho diseño puede determinar el éxito o el fracaso de un sistema de software.

3.1 Evaluación de la arquitectura de software

Es necesario tener en cuenta que la evaluación de una arquitectura, la cual involucra al equipo de evolución y a los *stakeholders*, no da un sí o un no, si es buena o mala, o una calificación de otra índole. Dicha evaluación expresa donde está el riesgo, es decir, las fortalezas y debilidades identificadas de la arquitectura del sistema y luego decidir si se prosigue con el desarrollo del sistema teniendo en cuenta las debilidades detectadas, si se refuerza la arquitectura o si hay que rediseñar la arquitectura del sistema.

Es recomendable realizar la evaluación de la arquitectura una vez especificada y no se haya iniciado su implementación, aunque la misma puede ser evaluada en cualquier momento de desarrollo.

Tradicionalmente se usan dos variantes para realizar la evaluación de la arquitectura teniendo en cuenta el avance en el que se encuentra el desarrollo del proyecto. Dichas variantes son la *evaluación temprana* y la *evaluación tardía*. En el primer caso no es necesaria la completa definición de la arquitectura ya que se pueden efectuar decisiones sobre esta a cualquier nivel. El segundo caso a diferencia del primero, se realiza cuando la arquitectura del sistema se encuentra definida y se ha terminado su implementación, es decir, en el momento de adquisición de un sistema ya implementado.

No obstante, también se debe tener en cuenta que la evaluación de la arquitectura de software debe realizarse en el momento justo en el que se cuenta con suficientes elementos para justificarla y la misma debe cumplir con dos criterios fundamentales:

- ✓ El sistema resultante cumple con los atributos de calidad.
- ✓ El sistema puede ser construido con los recursos disponibles, es decir, es construible.

En términos de coste, un buen momento para determinar cuándo realizar la evaluación podría ser cuando el equipo de desarrollo comienza a tomar decisiones que dependen de la arquitectura y que el costo de no tenerlas en cuenta es mayor que el costo de realizar una evaluación.

3.2 Técnicas de evaluación

Los atributos de calidad de un software están fuertemente influenciados por la arquitectura del sistema.

Bosch, 2000 [10].

Algunos de los atributos, los cuales no pueden ser medidos directamente sino a través de un análisis de la arquitectura, por los cuales puede ser evaluada una arquitectura son:

- ✓ **Disponibilidad:** Es la medida de disponibilidad del sistema para su posterior uso.

- ✓ **Desempeño:** Es el grado con el cual un sistema o componente cumple con sus funciones designadas, dentro de ciertas restricciones dadas, como velocidad, exactitud o uso de memoria.

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

- ✓ **Seguridad:** Es la mediada de la habilidad del sistema para resistir a intentos de uso no autorizados y negación de servicios, mientras se sirve a usuarios legítimos.

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

Existen varias técnicas que permiten realizar evaluaciones a la arquitectura y que se clasifican en cualitativas y cuantitativas.

En las técnicas de evaluación cualitativas se pueden utilizar escenarios, cuestionarios o listas de verificación, mientras que en las técnicas de evaluación cuantitativas se pueden emplear métricas, simulaciones, prototipos, experimentos o modelos matemáticos. La mayoría de los métodos de evaluación utilizan escenarios o secuencias específicas de pasos que involucran el uso o la modificación del sistema. Por lo regular, las técnicas de evaluación cualitativas son usadas cuando la arquitectura se encuentra en construcción *(evaluación temprana)*, mientras que las técnicas de evaluación cuantitativas, se usan cuando la arquitectura ya ha sido implantada *(evaluación tardía)*.

3.3 Método de evaluación de la arquitectura

Para realizar la estimación de la calidad a partir de la arquitectura, existen algunos métodos tales como *ATAM (Architecture Tradeoff Analysis Method), ARID (Active Reviews for Intermediate Design), SAAM (Software Architecture Analysis Method)*, etc., los mismos incorporan diferentes técnicas de evaluación y son utilizados frecuentemente.

Estos métodos en su mayoría se basan en escenarios y se usan para evaluar una arquitectura o para comparar varias. En el caso de *SAAM* es un método aconsejado para realizar la evaluación de arquitecturas donde el atributo de mayor peso es la modificabilidad. Por su parte el método *ARID* es recomendado para una arquitectura donde se profundiza más el atributo factibilidad. A diferencia de estos dos métodos anteriores, el *ATAM* se centra más en la arquitectura y los atributos.

La solución de arquitectura que se propone en este trabajo se encuentra especificada a alto nivel, es decir, un diseño poco detallado. Esta especificación a alto nivel impide la selección de escenarios que permitan validar si la arquitectura diseñada satisface a requisitos específicos. Como resultado de esto se realizará la evaluación atendiendo a cómo responden algunos elementos ante los principales atributos de calidad. Para ello se propone el método de evaluación *ATAM*. El mismo permite determinar el impacto de los atributos de calidad en el arquitectura, además provee un entendimiento interno, de cómo interactúan los principales objetivos de calidad. Cuando se evalúa usando *ATAM*, el objetivo es entender las consecuencias de las decisiones arquitecturales que respectan a los requerimientos de los atributos de calidad del sistema. Además permite determinar si los objetivos concebidos se pueden alcanzar por la arquitectura propuesta.

El método de análisis de arquitectura usado por *ATAM* permite que el análisis se pueda repetir. Además permite asegurar que las preguntas fundamentales respecto a los temas de arquitectura puedan ser respondidas tempranamente durante los requerimientos y los escenarios de diseño, cuando ocurre algún problema puede ser solucionado con bajos costos. Además provee a los *stakeholders* encontrar un conflicto y la solución en la arquitectura de software.

ATAM se centra fundamentalmente en evaluar las consecuencias de las decisiones arquitecturales tomadas en consideración a los atributos de calidad requeridos. *ATAM* es un método para identificar riesgos, esto significa que se detecta áreas de riesgos potenciales dentro de la arquitectura de un complejo sistema de software.

Esto tiene algunas implicaciones [13]:

- ✓ *ATAM* 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.
- ✓ *ATAM* producirá 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. (ej. latencia) puede ser el éxito. En cambio el éxito es lograr identificar las amenazas potenciales para el sistema.

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

Es sumamente importante en *ATAM* registrar cualquier riesgo, punto sensible y puntos de intercambio. Los riesgos son decisiones arquitecturalmente importantes, que no han sido tomadas (ej. El equipo de arquitectura no ha decidido cómo distribuir el tiempo o no ha decidido si usará base de datos relacional o base de datos Orientada a Objetos) o decisiones que han sido tomadas pero las consecuencias no han sido entendidas a plenitud (ej. el equipo de arquitectura ha decidido incluir una capa de portabilidad en el sistema operativo, pero no están seguro que función usar para acceder dentro de la capa).

Los puntos sensibles son parámetros en la arquitectura en los cuales la respuesta medible de algunos atributos de calidad son altamente correlacionados (ej. se puede determinar que la cantidad total de datos transmitidos en un sistema es altamente correlacionar con la cantidad de datos que se transmiten por un canal de comunicación en especifico).

Un punto de intercambio va a ser descubierto en la arquitectura cuando un parámetro de construcción arquitectural sea común para más de un punto sensible donde los puntos de calidad medibles son afectados indistintamente por cambio en el parámetro, (ej. si se aumenta la velocidad en el canal de comunicación mencionado anteriormente esto mejoraría en flujo de datos pero reduciría la confiabilidad, entonces la velocidad del canal es un punto de intercambio.)
Los pasos para aplicar el método son los que a continuación se relacionan:

Presentación:

1. *Presentar el ATAM:* Se describe el método a los *stakeholders*(Clientes representativos, Equipo de arquitectura, etc)
2. *Presentar la directriz del negocio:* El jefe del proyecto describe las metas del negocio y en qué lugares se debe centrar el esfuerzo, cuál va ser la directriz que guiará la arquitectura (ej. alta disponibilidad y alta seguridad).
3. *Presentar arquitectura:* El arquitecto describe la arquitectura propuesta, centrándose como va a estar direccionada hacia el negocio.

Investigación y análisis:

4. *Identificar estrategias arquitectónicas:* Estas son identificadas por el arquitecto pero no son analizadas.
5. *Generar árbol de utilidad de atributos de calidad:* Los factores de calidad que comprenden la utilidad del sistema (rendimiento, disponibilidad, seguridad modificabilidad, etc.) son capturados, especificados bajo un nivel de escenarios, comentados con estímulos, respuestas y priorizados.
6. *Analizar estrategias arquitectónicas:* Basada en los factores de mayores prioridad detectado en el paso 5, las estrategias arquitectónicas que guían esos factores son capturados y analizados (ej. una estrategia arquitectónica puede apuntar al entendimiento de las metas de rendimiento y esto puede ser subordinado a un análisis de rendimiento). Durante este paso son detectados los riesgos arquitectónicos, los puntos sensibles y los puntos de intercambio.

Pruebas:

7. *Tormenta de ideas y priorizar los escenarios:* Basado en los escenarios obtenidos en el árbol de utilidad del paso 5, se agranda el número de escenarios obtenidos del grupo de *stakeholders*. Este grupo de escenarios es priorizado mediante un proceso de votación que involucra a todo los *stakeholders*.
8. *Analizar estrategia arquitectónica:* Este paso repite el paso 6, pero aquí se resalta los escenarios obtenidos en el paso 7.

Reporte:

9. *Presentar resultados:*

En esta evaluación haciendo uso del método seleccionado se analizarán los siguientes atributos por los cuales será evaluada la calidad de la arquitectura propuesta:

- ✓ Rendimiento
- ✓ Portabilidad
- ✓ Modificabilidad
- ✓ Seguridad

3.4 Árbol de utilidades para la evaluación de la arquitectura propuesta.

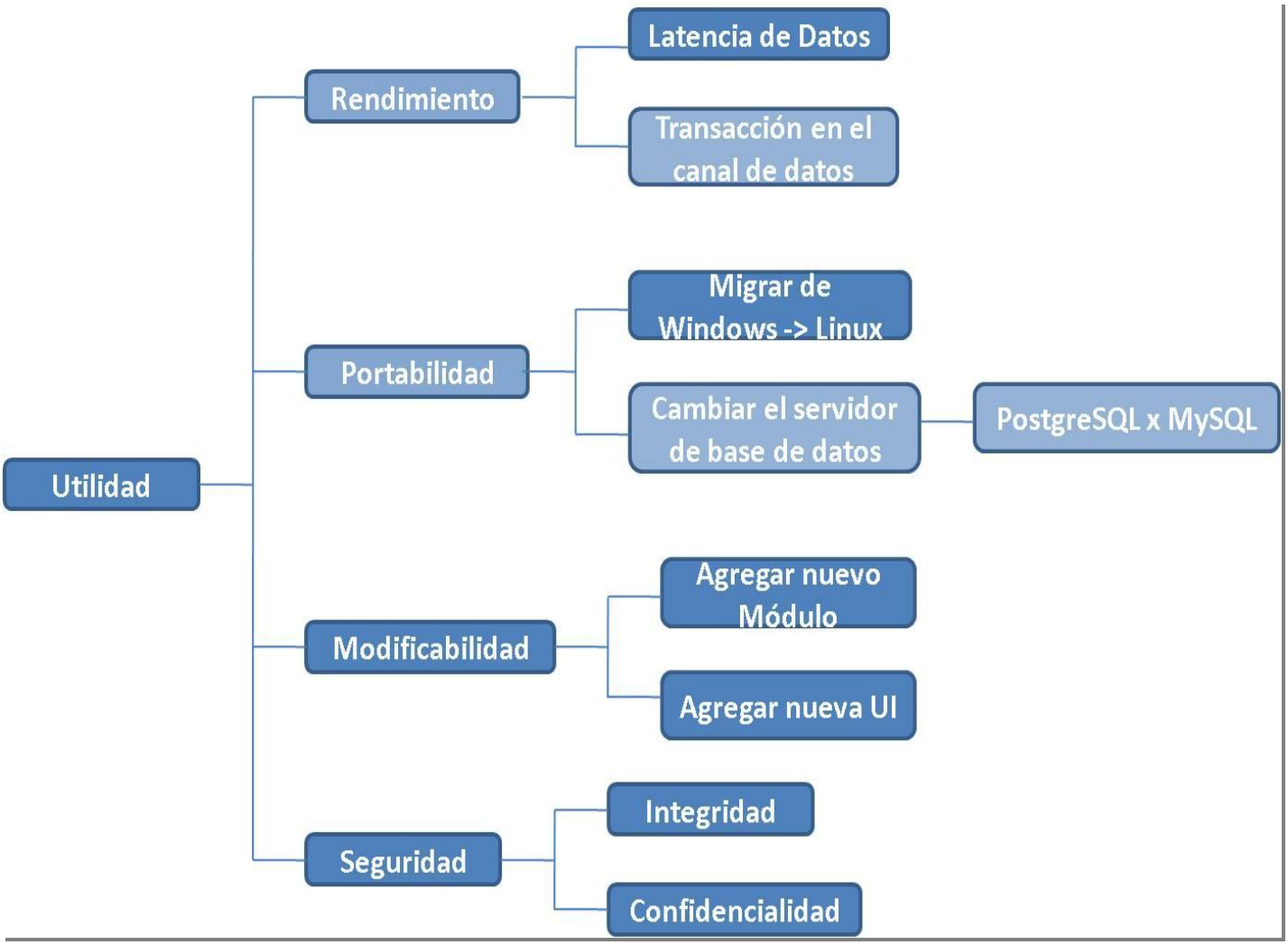


Fig. 22 Árbol de utilidades del método de evaluación de arquitectura aplicado. (El color más claro representa la debilidad)

Este árbol de utilidades es uno de los elementos de los cuales se vale el método *ATAM* para la realización de la evaluación de la arquitectura del sistema. Muestra también los principales atributos de calidad a través de los cuales se va a medir la calidad de dicha arquitectura y cómo se pueden ver afectados los mismos.

En el caso del atributo *Rendimiento*, el mismo se puede ver afectado en dependencia de cómo se comporte la latencia de los datos o la transmisión en el canal de los mismos, es decir, cómo se comporta el sistema a la hora de probar el flujo de información en el caso más crítico, que sería, el momento en que varios clientes simultáneamente se conectan y hacen uso del sistema o cuando se esté actualizando la base de datos. Estos momentos se definen como los más críticos por el hecho de que es cuando se trabaja con un alto volumen de datos en el canal de transacción.

El atributo *Portabilidad* se ve afectado cuando se quiere migrar de sistema operativo o cuando se quiera cambiar el gestor de bases de datos *PostgreSQL* por *MySQL*, *Oracle* u otro. En dependencia del valor que tome este atributo, se mostrará si la aplicación tiene la portabilidad como una fortaleza o una debilidad.

Para el caso del atributo *Modificabilidad* se puede decir que esta variable se vería afectada en dependencia de cómo se comporte el sistema o cuan flexible sea a la hora de agregar un nuevo módulo o una nueva interfaz de usuario.

El atributo *Seguridad* tomará valores en dependencia de la integridad de los datos que muestre la aplicación y la confiabilidad que el mismo sea capaz de transmitir. Esto dependerá de la asignación correcta o incorrecta de los permisos a los usuarios y de la restricción a determinadas áreas de información a las cuales los usuarios no tengan niveles de acceso garantizados.

3.4.1 Escenarios arquitectónicos para la aplicación del método ATAM.

Escenario	Cuando el controlador solicita información a la capa del modelo.
Objetivos del negocio	Procesar algún dato para obtenerlo de la base de datos.
Atributo de calidad	Mantenibilidad, portabilidad, modificabilidad, disponibilidad, flexibilidad.
Estímulo	Obtener una colección de objetos
Origen del estímulo	El gestor.
Elemento	Un método.
Ambiente	Se está realizando un proceso de búsqueda.
Respuesta	Se retorna una colección de objetos.
Medida de la respuesta	Nivel de abstracción, repercusión.
Preguntas	¿Cómo y cuáles son los componentes involucrados en la petición que se ven afectados?

Tabla 1. Escenario de solicitud de información al modelo.

Resultado obtenido de la ejecución del escenario:

La capa de funcionamiento y la de acceso a datos (Tabla 1) se comunican a través de objetos de *Propel*. En este escenario se le presta especial atención a tener el 100% de desacoplamiento entre estas capas puesto que la capa de datos es bastante crítica y propensa a cambios. Esta situación es fácil de resolver ya que *Propel* es un *ORM* que usa la librería *Creole* para lograr la abstracción total de la base de datos, cualquier cambio que se haga, como por ejemplo, cambiar el gestor de base de datos, sólo afectaría un fichero de configuración (*databases.yml*), si se cambia la estructura de la base de datos, también afectaría un solo archivo (*schema.yml*). En el caso de la aplicación *SIGIPE*, debido a que el archivo *schema.yml* fue generado automáticamente y a partir de la base de datos *SIGIPE*, la cual está montada sobre el gestor *PostgreSQL*, éste contiene ciertas particularidades propias del gestor en cuestión, resultando como consecuencia de esto que la aplicación no sea totalmente independiente del gestor, viéndose afectado el atributo *Portabilidad*, el cual ha sido identificado como una debilidad del sistema. Las razones por las cuales el archivo *schema.yml* fue generado de manera automática son: primeramente, porque generar este archivo de forma manual constituye un trabajo engorroso, también debido al gran número de tablas que componen la base de datos y porque aún no se cuenta con una herramienta automatizada capaz de hacer esta tarea.

Escenario	Cuando un módulo necesita comunicarse con otro para solicitar algún tipo de información o utilizar alguna funcionalidad.
Objetivos del negocio	Llevar a cabo una operación que se salga del dominio del módulo.
Atributo de calidad	Modificabilidad, flexibilidad, seguridad, integrabilidad.
Estímulo	Contabilizar.
Origen del estímulo	El gestor.
Elemento	Un método.
Ambiente	Se está llevando a cabo la contabilización de una operación contable. (p.e. contabilizar la cantidad de estudiantes de un proyecto determinado.)
Respuesta	Se registra el resultado de la contabilización.
Medida de la respuesta	Nivel de abstracción, repercusión, capacidad de incorporación de funcionalidades
Preguntas	¿Cómo compartir las funcionalidades y componentes y garantizar su seguridad en los módulos para llevar a cabo la operación? ¿Qué implica un cambio en alguna de estas funcionalidades o componentes? ¿Cuáles funcionalidades o componentes del módulo en que se está trabajando se ven afectados en el proceso?

Tabla 2. Escenario de comunicación entre módulos.

Resultado obtenido de la ejecución del escenario:

Dando respuestas a las interrogantes que surgen en este escenario, algunos de los componentes y funcionalidades comunes para varios módulos se encuentran en un nivel superior y otras dentro de los mismos. Estas funcionalidades y componentes están implementados de manera general pudiendo ser reutilizables. Un ejemplo de esto es cuando se necesita obtener un listado con un conjunto de datos de los estudiantes que pertenecen a cierto proyecto. En este caso, el módulo *Producción* necesita de una funcionalidad dentro del módulo *Estudiante* que devuelva un conjunto de datos necesarios para conformar la lista, siendo estos datos el nombre, el año académico y el sexo.

Continuando con este ejemplo, en caso de que hubiese que realizar cambios en la funcionalidad del módulo *Estudiante* porque se requiere que el resultado sea diferente al que se obtiene actualmente, sólo habría que modificar esta funcionalidad de manera que se ajuste a las necesidades de los módulos sin cambiar el código en los mismos. También es necesario tener en cuenta que varios módulos dependen del resultado de esta funcionalidad por lo cual la misma no se puede alterar por un beneficio particular de algún módulo. Estos argumentos dan la medida de cuan modificable puede ser el sistema ya que con sólo modificar una funcionalidad o componente no es necesario hacerlo para todos los módulos, identificándose así el atributo *Mdodificabilidad* como una de las fortalezas de la arquitectura propuesta para el sistema.

Este escenario también refleja que la seguridad es un punto importante a la hora de compartir las funcionalidades y componentes, ya que para tener acceso a los mismos se necesitan ciertas credenciales en dependencia del rol que desempeñen los usuarios. La seguridad y los permisos de acceso de los usuarios gestionarán a través de un modulo de administración. Este módulo será el encargado de asignar las credenciales para poder acceder a las funcionalidades dentro de los módulos. Esto permitirá tener la seguridad del sistema independiente de la aplicación, pudiéndose afectar los privilegios de acceso para cualquier componente del sistema y solo se afectaría el módulo de seguridad. Esta constituye una forma más de garantizar la *Modificabilidad* de la aplicación, así como la seguridad de la misma, siendo estos atributos potencialidades de la arquitectura.

Escenario	Se agrega un nuevo módulo para gestionar los datos del personal de soporte técnico de la facultad 6.
Objetivos del negocio	Agregar un nuevo módulo para comprobar luego el comportamiento del sistema.
Atributo de calidad	Modificabilidad, flexibilidad, escalabilidad, Integrabilidad.
Estímulo	Gestionar los datos del personal de soporte técnico.
Origen del estímulo	La aplicación.
Elemento	Un nuevo módulo
Ambiente	Se está incorporando un nuevo módulo para la gestionar datos.
Respuesta	Se crea el nuevo módulo con sus respectivas acciones y plantillas.
Medida de la respuesta	Nivel de aceptación del sistema ante la incorporación de un nuevo módulo.
Preguntas	¿Cómo garantizar la seguridad de este nuevo módulo? ¿Cómo integrar este nuevo módulo?

Tabla 3. Escenario de creación de un nuevo módulo.

Resultado obtenido de la ejecución del escenario:

A través de este escenario se refleja el nivel de *Modificabilidad* y *Escalabilidad* que posee el sistema como resultado de la arquitectura definida, ya que al agregar un nuevo módulo la aplicación no se ve afectada de forma negativa.

Para garantizar la seguridad de este nuevo módulo se crearán las restricciones correspondientes para garantizar el acceso de los usuarios a las áreas definidas para los mismos. En este caso la seguridad se expande desde la seguridad ya implementada hasta este nuevo módulo re-utilizando tanto las credenciales como los roles y permisos ya previamente definidos. La integración de este nuevo módulo se garantiza incorporando sus funcionalidades en el menú general de la aplicación, mostrándosele a los usuarios solo aquellas funcionalidades a las que tienen acceso en dependencia de la credencial que posean. Este escenario también muestra la *Modificabilidad* y *Seguridad* como fortalezas de la arquitectura propuesta para el sistema.

3.5 Conclusiones

En este capítulo realizó un estudio y se plantearon las técnicas y métodos a utilizar para llevar a cabo la evaluación de una arquitectura.

Se definió el método *ATAM* para realizar la evaluación de la arquitectura del sistema para la gestión de la información de la facultad 6.

A partir de todos los elementos que se han reflejado como producto de aplicar el método *ATAM* haciendo uso del árbol de utilidades y también de los escenarios de evaluación, se puede decir que las fortalezas residen en la *Modificabilidad* y *Seguridad* del sistema, además de que la solución propuesta es totalmente flexible, reparable y mantenible.

La aplicación del método permitió identificar como puntos débiles la baja *portabilidad* a la hora de cambiar el gestor de bases de datos y el medianamente bajo *rendimiento* que ofrece el mismo ante situaciones de carga en el canal de transacciones de datos.

Es necesario aclarar que la carga en el canal de transacciones está relacionada las actualizaciones, ya que en las situaciones normales de manejo de los datos el sistema se comporta de manera satisfactoria.

Conclusiones generales

Con el desarrollo del presente trabajo, centrado en la propuesta de un diseño arquitectónico adecuado para el *Sistema para la Gestión de Información de Profesores y Estudiantes de la Facultad 6 (SIGIPE)*, se obtuvieron resultados concretos que muestran el cumplimiento tanto del objetivo general como de los objetivos específicos propuestos para la investigación.

- ✓ Se identificaron los patrones y estilos arquitectónicos a utilizar para el desarrollo del sistema. Estos patrones son los que con un correcto uso garantizan una robusta arquitectura.
- ✓ Se definieron las herramientas y tecnologías más convenientes para el desarrollo de la aplicación, principalmente aquellas que pertenecen al software libre.
- ✓ Se elaboró el documento de la arquitectura donde entre otras se reflejaron las distintas vistas que propone la metodología *RUP*, así como los estándares de codificación y las métricas y restricciones del diseño.
- ✓ Se validó la arquitectura mediante el método *ATAM* y de esto resultó que es factible utilizar la arquitectura propuesta.
- ✓ Se contribuyó con la integración de los módulos los cuales contienen toda la información relacionada con los estudiantes y profesores de la facultad 6.

Recomendaciones

Con el propósito de ampliar y mejorar la documentación de la arquitectura presentada en este trabajo, se plantean las siguientes recomendaciones:

- ✓ Se recomienda la continuidad de estudio del framework *Symfony* debido a la evolución constante que presenta el mismo, así como el seguimiento de las nuevas versiones del framework en caso de que se desee liberar nuevas versiones del proyecto.
- ✓ Se recomienda evaluar la posibilidad de generar el archivo *schema.yml* de tal manera que la aplicación pueda ser independiente de cualquier gestor que se desee usar y que el mismo sea soportado por el framework *Symfony*.
- ✓ Se recomienda tener el servidor de aplicaciones independiente del servidor de base de datos según se definió para el despliegue. Si por cuestiones de falta de recursos u otras no es posible mantener la independencia de estos servidores, los mismos se pueden montar en el mismo ordenador sin problema alguno que no sea la carga de este, pudiendo verse afectado en un nivel mayor el rendimiento del sistema.

Referencias bibliográficas

[1] Kruchten, Philippe. Architectural Blueprints--The 4+1 View Model of Software Architecture. IEEE Software, Institute of Electrical and Electronics Engineers (November 1995, pp. 42-50).

[2] Carlos Billy Reynoso. Introducción a la Arquitectura de Software (1996).

[3] Standard IEEE 1471-2000

[4] Taylor y Medvidovic [TMA+95]

[5] Christopher Alexander. A Pattern Language: Towns, Buildings, Construction (1977).

[6] Carlos Bily Reynoso. Introducción a la Arquitectura de Software (Marzo 2004).

[7] Microsoft. Patterns & Practices (Octubre 2004).

[8] Carlos Reynoso – Nicolás Kicillof. Lenguajes de Descripción de Arquitectura (2004).

[9] Rick Kazman, Philippe Kruchten. "Integrating Software-Architecture-centric methods into the Rational Unified Process", *CMU/SEI-2004-TR-011(2004).*

[10] Jan Bosch (Bosch 2000).

[11] Clements, P. A survey of Architecture Description Languages (1996).

[12] Steve Vestal (ves 1993).

[13] Kazman, R., M. Klein. ATAM: Method for Architecture Evaluation (2000).

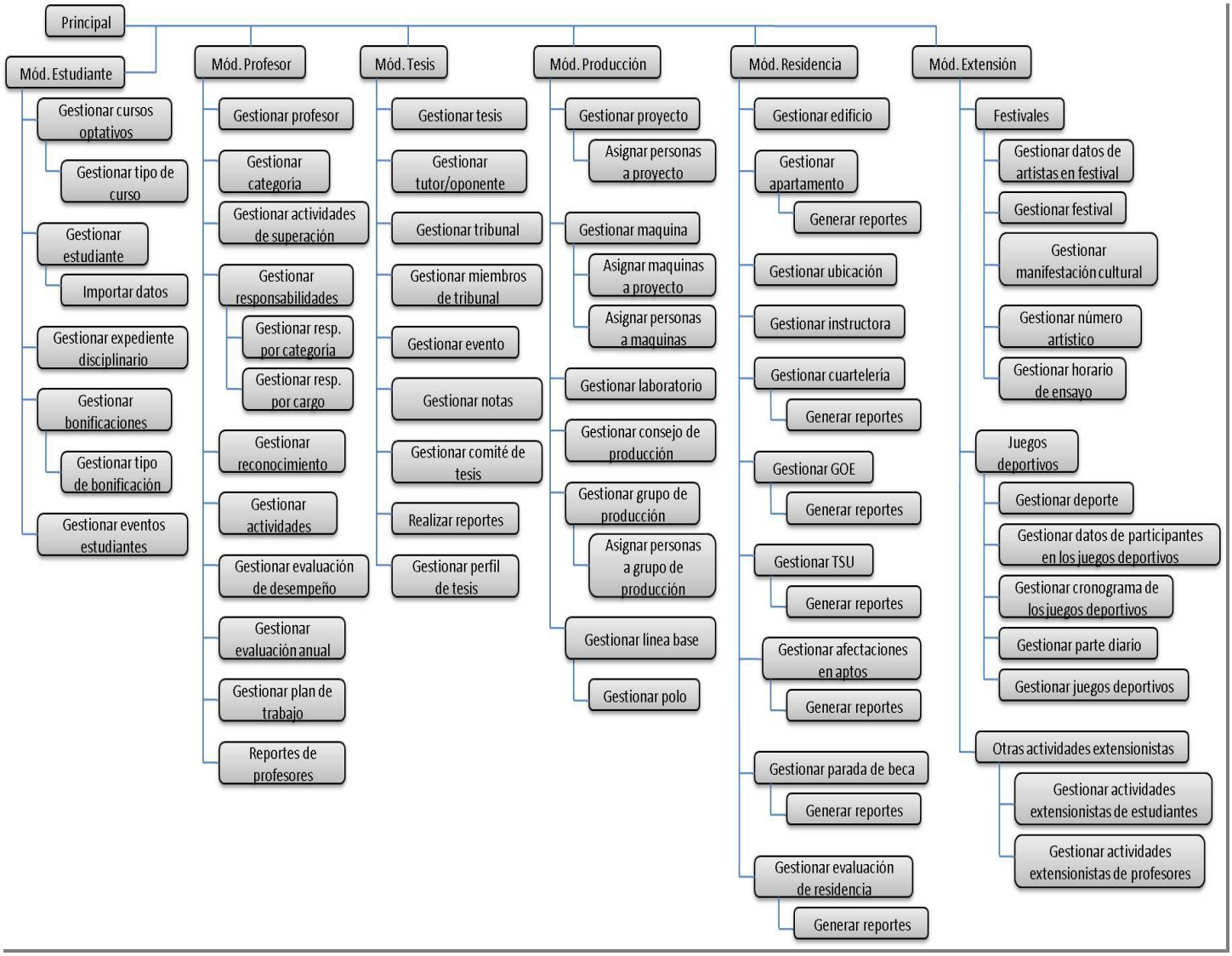
Bibliografía

1. Guía definitiva de Symfony.
2. http://www.Symfony-project.org/installation/1_2/
3. http://www.programacion.com/php/articulo/bbdd_disenyo/
4. CELIS Ismael “El ataque de los Frameworks”. Disponible en: http://www.estadobeta.com/2005/11/20/el-ataque-de-los-frameworks/ (15/03/2008).
5. 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).
6. 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.
7. http://www.computer.org/portal/cms_docs_ieeecs/ieeecs/images/IBM_Rational/FINAL.SW.V12N6.42.pdf.
8. 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 16 de Junio de 2008].
9. MICROSOFT CORPORATION. What Architect Job Roles Are Recognized By the
10. Microsoft Certified Architect Program? <http://www.microsoft.com/learning/mcp/architect/specialties/default.mspx> [citado 11 de Julio de 2007] .
11. IBM SOFTWARE GROUP. Characteristics of a Software Architect http://www.ibm.com/developerworks/rational/library/mar06/eeles/> [citado 15 marzo 2006].
12. BREDEMEYER. Architect Competency Framework http://www.bredemeyer.com/pdf_files/ArchitectCompetencyFramework.PDF> [citado 16 de Junio de 2008].
13. http://www.apache.org/
14. 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>.
15. Documentación del SEI en la universidad Carnegie Mellon <http://www.sei.cmu.edu/publications/publications.html>.

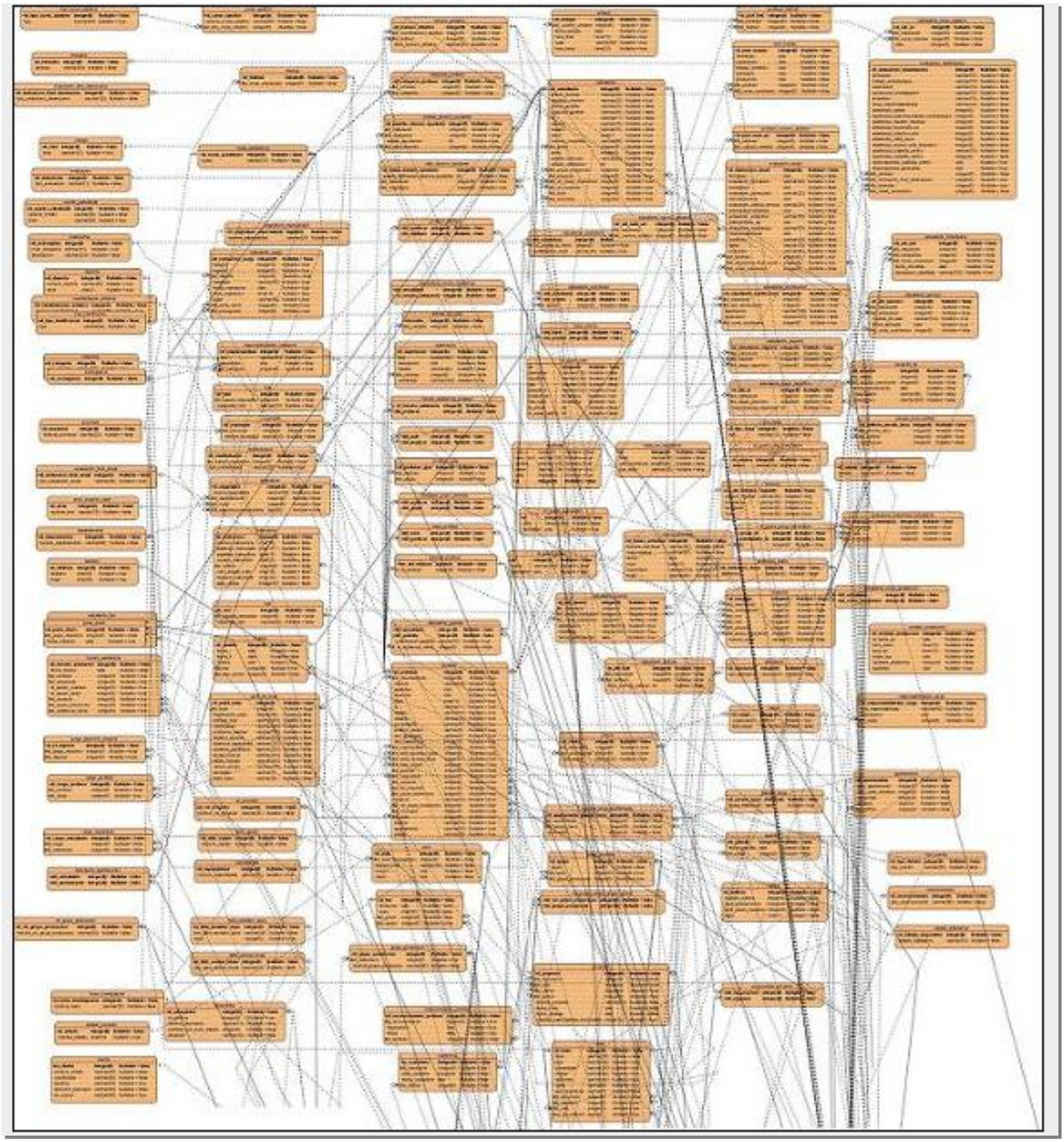
16. Booch, G., Rumbaugh, J., Jacobson, I. “El Proceso Unificado de Desarrollo de Software”. Addison-Wesley Longman, 2000.
17. Booch, G., Rumbaugh, J., Jacobson, I. “El Lenguaje Unificado de Modelado”, Addison-Wesley, 2000.
18. Mendoza Sánchez, María A. “Metodologías de desarrollo de software”. Disponible en:http://www.informatizate.net/articulos/metodologias_de_desarrollo_de_software_07062004.html.
19. Kruchten, P. “The Rational Unified Process: An Introduction”, Addison Wesley, 2000.
20. Sitio oficial del Visual Paradigm. Disponible en: http://www.visual-paradigm.com/product/vpuml.
21. 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/architecture/ata_method.html
http://www.sei.cmu.edu/pub/documents/96.reports/pdf/tr036.96.pdf

Anexos

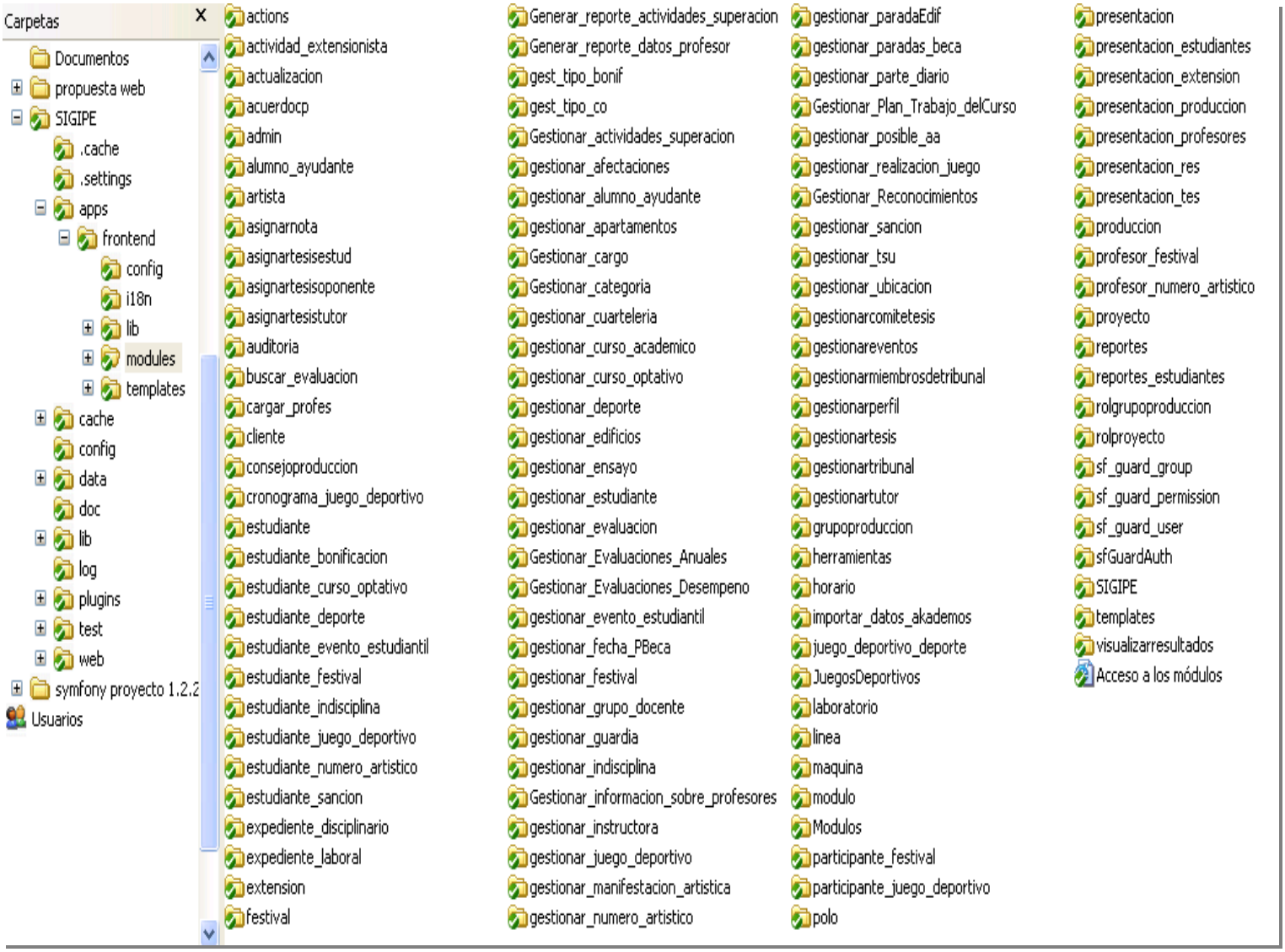
Anexo 1: Mapa de navegación.



Anexo 2: Modelo de datos del sistema.



Anexo 3: Vista del árbol de directorio de la aplicación SIGIPE desde el explorador de Windows. A la izquierda la estructura de las carpetas, a la derecha los módulos del sistema.



Glosario de términos

SIGIPE: Sistema para la Gestión de la Información de Profesores y Estudiantes de la Facultad 6.

UCI: Universidad de las Ciencias Informáticas.

MVC: Modelo Vista Controlador.

PHP: Personal Home Page.

RUP: Rational Unified Process.

UML: Unified Modeling Language.

CVS: Current Versions System.

SVN: SubVersion.

SQL: Structured Query Language.

GB: Gigabits

TB: Terabits

IDE: Integrated Development Environment.

POO: Programación Orientada a Objetos.

HTML: Hypertext Markup Language.

DHTLM: Dynamic HTML.

HTTP: Hypertext Transfer Protocol.

XML: Extensible Markup Language.

MSF: Microsoft Framework Solutions.

ADL: Architectural Describing Language.

ATAM: Architecture Tradeoff Analysis Method.

ARD: Active Review Design.

ARID: Active Reviews for Intermediate Design.

SAAM: Software Architecture Analysis Method.