Universidad de las Ciencias Informáticas

Facultad 3

Título: “Propuesta de un Lenguaje de Descripción de Arquitectura para los proyectos productivos de la Facultad 3”

Trabajo de Diploma para optar por el título de

Ingeniero en Ciencias Informáticas.

Autores: Omel Mieres Durán

Yandry Acuña Rivera

Tutor: Ing. Yunieski Fábregas Santos

Co-Tutor: Ing. Miguel Ángel Coppén Golpe

Ciudad de la Habana, Junio del 2008

Declaración de Autoría.

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

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

Yandry Acuña Rivera **Omel Mieres Durán**

_____ _____

Firma del Autor Firma del Autor

Ing. Yunieski Fábregas Santos

_____

Firma del Tutor

Agradecimientos

Queremos agradecer en primer lugar a nuestros familiares por todos los esfuerzos realizados para que este día llegara.

A la genial idea de nuestro Comandante Fidel, que nos permitió formar parte de esta Tropa de Futuro.

A todos los profes que nos brindaron sus enseñanzas y experiencia a lo largo de nuestras vidas como estudiantes.

A nuestros queridos profes de la UCI, en especial a Rudel, Coppén y Pascual.

A todos los que pusieron su granito de arena para que este trabajo saliera adelante. (Nuestro tutor Yunieski, Coppén, Rudel, Jose, Adolfito, Henryk, Yunier, Olivia, José Raúl).

A los miembros del Proyecto de RN por apoyarnos en todo lo que les fue posible, en especial a Lurdes (El perrito no está tan feo nada).

A todos los que en algún momento se preocuparon por cómo nos iba con la tesis, esas cosas son muy importantes.

A nuestros amigos que siempre han estado pendientes de nosotros y a nuestro lado en todo momento.

Quisiéramos que no se nos quede nadie, por eso le agradecemos a todo el mundo, de ese sí que nadie se escapa.

Finalmente a la vida, por permitirnos llegar a este momento.

Muchas gracias

Omel y Yandry.

Dedicatoria

A mi mami linda, gracias por las fuerzas que me diste siempre, por tu ejemplo, por ser como eres, por no rendirte nunca, por ser especial, por quererme tanto, sin tus consejos no lo hubiese logrado, eres mi vida. A mi papito del alma, tal vez no seas perfecto, pero no puede haber un papá mejor, lo aseguro, te quiero mucho. A dos personas que no tienen defectos, que son todo cariño y amor, que viven para mí, mis dos abuelitas Yaya y Mirta, las amo. A la niña de mis ojos, a esa que es mi prima, mi hermana, mi niña chiquitica, mi todo, lo he logrado entre otras cosas, porque sé que soy tu guía. A esos tíos maravillosos, casi padres que tengo, ustedes saben que han sido muy importantes (Tata, Omy y Toto), al fin lo logramos, a otros tíos que quiero y admiro mucho (Yosy, Jorge, Jesús, Ramoncito's, Alina, Evelito), a mis primos, y el resto de esa familia tan unida y linda que tengo. A mis vecinos que son mucho más que eso (Dany, Liset y mis 2 sobrinitos Danilo y Daniel, a Lurdes, Carli y Luli). Mi mamá no me dio hermanos, pero la vida me premió con muchos, a ustedes mis hermanos desde más pequeño (Hamlet, Migue, Darién, mi Nené, Carli, Robe y Nesti) y a los de aquí que se que también estarán para siempre (Ome, Johnnyto, Shalymar, Samu, Juli, Coppén, Yunie, Ore, el Pesca, el Mene, el Fide, Mary, Suse). Al Piquete de la Pegada, a mis compañeros de aula del IPVCE, a mis amigos y compañeros de aula de aquí de la UCI especialmente a (Iosme, Carli, el Socito), a mis 2 equipos de volley, el UCI y el de la Facultad, a la gente del Basket, a mis hijos de 1ero, a todas mis amiguitas que las quiero mucho y a los letales del edificio 11, como nos vamos a extrañar…

Los quiere Yandry.

Dedicatoria

A mí mamita querida por consentirme tanto y hacer siempre lo mejor para mi sin importar las consecuencias, por estar siempre a mi lado y guiarme por los mejores caminos de la vida.

A mí hermano que siempre ha sido más que mi hermano, mi hermano, mi amigo, mi compañero en los momentos más difíciles.

A mí querida familia que está al tanto de mi vida, aconsejándome y guiándome por un buen camino.

A todos mis compañeros de aula que desde primer año supimos imponernos a todas las adversidades y ya al cabo de 5 años nos tenemos que separar, pero siempre quedarán esos momentos que pasamos juntos y que ni el tiempo ni la distancia, ni el tiempo podrán borrar.

A mis hermanos, Coppén, Yandry, Johnny, Samuel, Julito y Shalymar, por hacer mucho más agradable mi vida en esta universidad y estar en las buenas y en las malas.

A todas aquellas personas que ayudaron a hacer realidad mi sueño de ser un ingeniero y una mejor persona después de estos 5 años.

A todos gracias.

Con mucho cariño Omelito.

Resumen

Las decisiones arquitectónicas son fundamentales en el éxito de una aplicación. En la actualidad los sistemas de software requieren la combinación de diferentes tecnologías, así como de plataformas de hardware y software para llegar a cumplir dichas necesidades. El presente trabajo contiene un estudio de los principales elementos que constituyen la Arquitectura de Software con el propósito de lograr una organización estructural del Sistema de Gestión de Inventario y Almacén. Resulta primordial definir las características fundamentales de las tecnologías y aspectos fundamentales del diseño, tratando de transmitir una idea clara y objetiva utilizando la relación entre sus componentes, conectores y restricciones y a través de los artefactos que resultan más representativos según Acme Studio. La Arquitectura de un sistema debe ser modular, flexible y cumplir con los parámetros de funcionalidad, eficiencia y confiabilidad. Además, contiene un análisis de los resultados obtenidos a partir de los objetivos planteados y las restricciones arquitectónicas que se pueden encontrar evitando que pueda existir un mal funcionamiento de la solución en un momento determinado.

Palabras Claves: Arquitectura de Software, Sistema de Gestión de Inventario y Almacén, Acme Studio, ADL, Componentes, Conectores, Escenarios, Patrones**,** Vistas., Modular, Flexible.

Índice

Índice de Tablas

Índice de Figuras

Introducción

El mundo actual se encuentra en un estado de transición, donde la era industrial va dando espacio a una nueva, caracterizada por el uso de la información y el conocimiento, nueva era en la cual la actividad humana muestra un gran nivel de desarrollo basado en el conocimiento y las Tecnologías de la Información y las Comunicaciones (TIC), las cuales cada vez muestran un mayor avance con el objetivo de obtener más y mejores programas que hacen más cómodo el uso de las mismas, estableciendo y creando nuevos mercados y oportunidades económicas.

En la actualidad la industria del software se abre paso en una carrera donde muchos países, instituciones y empresas privadas ven una gran fuente de recursos y luchan por lograr un desarrollo del conocimiento y una respetada posición en el mercado.

Cuba a pesar de ser un país bloqueado y subdesarrollado va a continuar con su estrategia de formar capital humano, ahora tomando parte en esta carrera, proponiéndose garantizar el sustento de la economía a través de la industria del software.

Para ello se han creado algunos mecanismos e instituciones para que la producción de software se eleve, una de estas instituciones es la Universidad de las Ciencias Informáticas (UCI), surgida en el año 2002 en medio de la Batalla de Ideas, con el colosal propósito de convertirse en un pilar fundamental dentro del desarrollo de software en el país.

La UCI, para lograr un mejor desarrollo productivo, y cumplir con el objetivo fundamental por el que fue creada, se ha dividido en diez facultades, cada una de estas con un perfil de trabajo diferente, pero orientados todos al desarrollo informático. Estas facultades han sido el espacio donde la producción de software ha alcanzado un auge considerable y se ha convertido en la actividad esencial, siendo una necesidad el perfeccionamiento de los proyectos productivos que en ellas se desarrollan. Para el desarrollo de estos proyectos se utilizan diferentes lenguajes y herramientas que junto al capital humano garantizan la calidad y soporte del producto.

En la actualidad debido al desconocimiento sobre los ADL[1], en la facultad se utiliza UML como Lenguaje de Descripción de Arquitectura lo que afecta la productividad y calidad del proceso de desarrollo de software debido a que este lenguaje es una notación para especificar, visualizar y documentar sistemas de software desde el punto de vista de diseño, lo cual hace que se pierdan

[1] Lenguaje de Descripción de Arquitectura.

cualidades necesarias en la descripción arquitectónica como la abstracción, además de mecanismos para descomponer un sistema en componentes y conectores que mejoran la comprensión y el entendimiento con el equipo de desarrollo.

A raíz de esta situación problemática se identifica como **problema científico** la siguiente interrogante: ¿Cómo obtener un nivel de abstracción arquitectónico para agilizar el proceso de arquitectura de software en la Facultad 3?

El **objeto de estudio** del Trabajo de Diploma es el Proceso de Arquitectura en el desarrollo de Software, teniendo como **campo de acción** los ADL en el Proceso de Arquitectura de Software de la Facultad 3.

El **objetivo general** que se persigue con el presente trabajo es: Proponer un Lenguaje de Descripción de Arquitectura (ADL) para la agilización del proceso de arquitectura de software en los proyectos productivos de la Facultad 3 y su puesta en práctica mediante un caso de estudio.

Como **objetivos específicos** se identifican los siguientes:

- Comprobar la superioridad de los ADL sobre el UML para la descripción de arquitecturas de software con un nivel alto de complejidad desde el punto de vista estructural.
- Proponer un ADL lo más completo posible para ser utilizado por la gran mayoría de los proyectos de la facultad.
- Describir los patrones y frameworks utilizados en el sistema así como su implantación en el mismo.
- Representar la Arquitectura definida para el Sistema Gestión de Inventario y Almacén utilizando el ADL propuesto.

Hipótesis: Con la utilización de un ADL para la realización de la Descripción de la Arquitectura de los proyectos de la Facultad 3 se logrará un nivel de abstracción ventajoso desde el punto de vista arquitectónico, propiciando esto la agilización del proceso de desarrollo.

Para dar cumplimiento a cada uno de estos objetivos se realizarán las siguientes **tareas:**

- Sistematización del estudio del estado del arte.
- Realización de un estudio sobre las principales tareas a realizar por el rol de Arquitecto dentro de un equipo de desarrollo de software.
- Determinación de un ADL a partir del estudio de las características y ventajas principales de varios de ellos.

- Descripción de la Arquitectura del Sistema Gestión de Inventario y Almacén a partir del ADL determinado.
- Solución a las situaciones encontradas durante el desarrollo de la tarea anterior.
- Validación mediante la descripción de la Arquitectura del Sistema Gestión de Inventario y Almacén con la utilización del ADL propuesto.

Resultados Esperados:

Los resultados que se esperan obtener con la realización del presente trabajo son los siguientes:

- Evidenciar la necesidad de utilizar los ADL para la Descripción de la Arquitectura en los proyectos productivos de la Facultad 3.
- Un ejemplo de Descripción Arquitectónica utilizando un ADL.

Estructura del Documento:

Capitulo 1: Fundamentación Teórica.

Definición del marco teórico y del modelo teórico de la investigación, estudio del arte de la Arquitectura de Software y del rol de arquitecto.

Capitulo 2: Propuesta de ADL para la Facultad 3.

En este capítulo se realiza un estudio profundo sobre la necesidad de utilizar los ADL con vistas a optimizar el proceso de desarrollo de software, además se muestran algunos de los ADL más utilizados en la actualidad, haciendo énfasis en sus principales características y ventajas. Además se propone una guía para la utilización de los ADL en la Facultad.

Capitulo 3: Aplicación del ADL propuesto al Sistema Gestión de Inventario y Almacén.

En este capítulo se hace una propuesta de solución al problema planteado en el diseño teórico de la investigación. Contiene la descripción del documento Línea Base de la Arquitectura del Sistema de Gestión de Inventario y Almacén.

La solución propuesta está dividida en dos sub-tópicos principales:

- Línea Base de la Arquitectura
- Documento Descripción de la Arquitectura

Capítulo 1: Fundamentación Teórica.

1.1 Introducción.

A esta altura del desarrollo de la arquitectura de software, podría pensarse que hay abundancia de herramientas de modelado que facilitan la especificación de desarrollos basados en principios arquitectónicos, que dichas herramientas han sido evaluadas y estandarizadas hace tiempo y que son de propósito general, adaptables a soluciones de cualquier mercado vertical y a cualquier estilo arquitectónico, la situación como se verá en este trabajo de diploma es mucho más compleja, ya que de la Arquitectura de Software dependen la mayoría de los elementos que intervienen en el desarrollo del sistema que se está implementando. En este capítulo se realiza un análisis del estado del arte de la arquitectura de software, sus distintos enfoques y tendencias actuales para modelar el diseño teórico sobre el que se fundamenta la investigación.

1.2 Ingeniería basada en componentes.

La Ingeniería de Software basada en Componentes (Component-Based Software Engineering, CBSE) está enmarcada en la Ingeniería de Software y existe un interés cada vez mayor en ella.

Este interés está dado primeramente por la nueva concepción de desarrollo de software de forma industrial, el desarrollo de las factorías de software (1) y la concepción de que es mucho más fácil y rápido reutilizar componentes ya desarrollados y probados, a tener que implementarlos desde el principio.

Los antecedentes más significativos de las plataformas de componentes se pueden establecer en Ambiente Distribuido de Computación (Distributed Computing Environment (DCE)) y (Common Object Request Broker Architecture (CORBA)), desarrolladas por iniciativa de los consorcios de la Fundación de Software Libre (OSF) y la Dirección de Grupo de Objetos (OMG), respectivamente. Aunque DCE (2) tuvo en un principio una buena acogida por parte de la industria, algunas de las limitaciones que presentaba (no era orientada a objetos, no definía servicios de transacciones y solo permitía invocaciones estáticas, no dinámicas) le hizo perder gran parte de su cuota de mercado en el desarrollo de aplicaciones distribuidas. Sin embargo, la especificación de la arquitectura CORBA, (3), (4), a pesar de definirse como una plataforma de objetos distribuidos, asienta los principios básicos de la tecnología de componentes,

habiendo consolidado en gran medida su posición. Posteriormente han aparecido diversas plataformas, entre las que debemos citar, por su importancia, Modelo de Objetos de Componentes (COM), desarrollada por Microsoft (5), (6) y JavaBeans, desarrollada por Sun (7).

Estas tres plataformas constituyen los estándares de facto en este terreno, aunque su continua evolución ha dado lugar a numerosos modelos y productos derivados, entre los que se encuentran Modelo de Componentes CORBA (CCM), EJB (Enterprise Java Beans), DCOM (Distributed COM) (8).

Uno de los campos en los que la tecnología de componentes se ha mostrado más activa es en el desarrollo de marcos de trabajo (Frameworks) (9). Estos se definen como un diseño reutilizable de todo o parte de un sistema, representado por un conjunto de componentes abstractos, y la forma en la que dichos componentes interactúan.

Un framework se considera como la plantilla de la aplicación o de la parte de la aplicación que se quiera desarrollar, debe ser bien seleccionado en dependencia de las restricciones del sistema y de las posibilidades que brinde el framework, estos a la vez no son excluibles sino que se pueden extender y combinarse para lograr una interacción entre distintas partes de un sistema.

1.3 Arquitectura de software.

En el desarrollo de un sistema o proyecto de software el arquitecto juega un papel fundamental. Este una vez que conoce los requisitos con que debe cumplir el software, debe trazarse una estrategia y articular los patrones que se podrán utilizar para modelarlo, aplicando una convención gráfica o algún lenguaje avanzado de alto nivel de abstracción.

En la actualidad podría pensarse que abundan las herramientas de modelado que hacen más fácil la especificación de desarrollos basados en principios arquitectónicos. En ocasiones se cometen errores al modelar sistemas desde el punto de vista arquitectónico pues se asemeja al modelado en ambientes ricos en prestaciones gráficas, como es el caso del modelado de tipo CASE (Ingeniería de Software Asistida por Computadores) o Lenguaje Unificado de Modelado (UML). En la década de 1990 y en lo que va del siglo XXI, se han materializado diversas propuestas para describir y razonar en términos de arquitectura de software; muchas de ellas han asumido la forma de Lenguajes de Descripción de Arquitectura (ADLs). Estos suministran construcciones para especificar abstracciones arquitectónicas y

mecanismos para descomponer un sistema en componentes y conectores, especificando de qué manera estos elementos se combinan para formar configuraciones y definiendo familias de arquitecturas o estilos. Contando con un ADL, un arquitecto puede razonar sobre las propiedades del sistema con precisión, pero a un nivel de abstracción convenientemente genérico. Algunas de esas propiedades podrían ser, por ejemplo, protocolos de interacción, anchos de banda y latencia, localización del almacenamiento, conformidad con estándares arquitectónicos y previsiones de evolución ulterior del sistema.

Las definiciones clásicas de la arquitectura la definen como:

Mary Shaw y David Garlan, sugieren que la arquitectura del software sea un nivel del diseño referido a las ediciones "…más allá de las estructuras de los algoritmos y de datos del cómputo; diseñar y especificar la estructura del sistema total emerge como nueva clase de problema. Las ediciones estructurales incluyen la organización gruesa y la estructura global del control; los protocolos para la comunicación, la sincronización, y el acceso a los datos; asignación de la funcionalidad para diseñar elementos; distribución física; composición de los elementos del diseño; escalamiento y funcionamiento; y selección entre alternativas del diseño" (10).

En esta definición no queda claro donde se debe llevar a cabo la arquitectura en el ciclo de desarrollo del software pues no define si comienza en un momento entre la definición de los requisitos funcionales y el diseño del sistema.

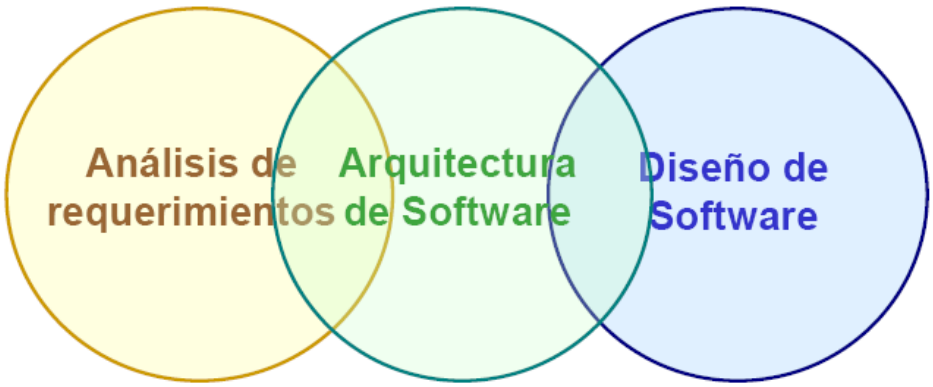

Figura 1. Ubicación de la Arquitectura de Software dentro del Proceso de Desarrollo de Software.

Capítulo 1

En el año 2000 el Instituto de Ingenieros Eléctricos y Electrónicos (IEEE) hace la definición oficial de Arquitectura de Software (AS) en su documento IEEE 1471, que reza así:

"La Arquitectura de Software es la organización fundamental de un sistema encarnada en sus componentes, las relaciones entre ellos y el ambiente y los principios que orientan su diseño y evolución".

Este documento de la IEEE 1471 ha sido adoptado por todas las organizaciones y empresas relacionadas con el desarrollo de software, donde la AS tiene gran importancia.

Esta definición, deja claro que la AS no se inscribe en ninguna metodología de desarrollo, sino que es en sí, una disciplina que se desarrolla durante todo el ciclo de desarrollo de software.

Existen otras definiciones clásicas de la arquitectura tales como:

"…más allá de las estructuras de los algoritmos y de datos del cómputo; diseñar y especificar la estructura del sistema total emerge como nueva clase de problema. Las ediciones estructurales incluyen la organización gruesa y la estructura global del control; los protocolos para la comunicación, la sincronización, y el acceso a los datos; asignación de la funcionalidad para diseñar elementos; distribución física; composición de los elementos del diseño; escalamiento y funcionamiento; y selección entre alternativas del diseño" (10).

El Proceso Unificado de Modelado (RUP) plantea:

"Una arquitectura es el sistema de decisiones significativas sobre la organización de un sistema de software, la selección de los elementos estructurales y de sus interfaces por los cuales el sistema es compuesto, junto con su comportamiento según lo especificado en las colaboraciones entre estos elementos, la composición de estos elementos estructurales y del comportamiento en subsistemas progresivamente más grandes y el estilo arquitectónico que dirigen esta organización, los elementos y sus interfaces, sus colaboraciones y su composición" (11).

Esta define la AS como una etapa más de la Ingeniería de Software, da una visión de la arquitectura como algo más que la construcción de herramientas o de tecnologías a usar en el desarrollo del sistema. Está enfocada a la orientación a objetos y a UML, según lo plantea la metodología de desarrollo RUP.

1.4 Rol del arquitecto.

El Arquitecto de Software debe dominar la mayor cantidad de tecnologías de software y prácticas de diseño, para así poder tomar decisiones adecuadas para garantizar el mejor desempeño de las aplicaciones. El rol del arquitecto de software es crítico y sumamente importante, puesto que requiere de una gran variedad de conocimientos, tales como: ingeniería de requerimientos, teoría de arquitecturas de software, codificación, tecnologías de desarrollo, plataformas de hardware y software.

De igual manera, requiere de saber negociar intereses encontrados de múltiples involucrados en el desarrollo de un sistema de software; promover la colaboración entre el equipo; entender la relación entre atributos de calidad y estructuras; ser capaz de transmitir claramente la arquitectura a los equipos; escuchar, y entender múltiples puntos de vista. El arquitecto de software debe interaccionar con todos los involucrados en el desarrollo de un sistema de software, y ser capaz de dialogar con el analista para obtener los requerimientos significativos, diseñarlos y transmitirlos al programador para su codificación.

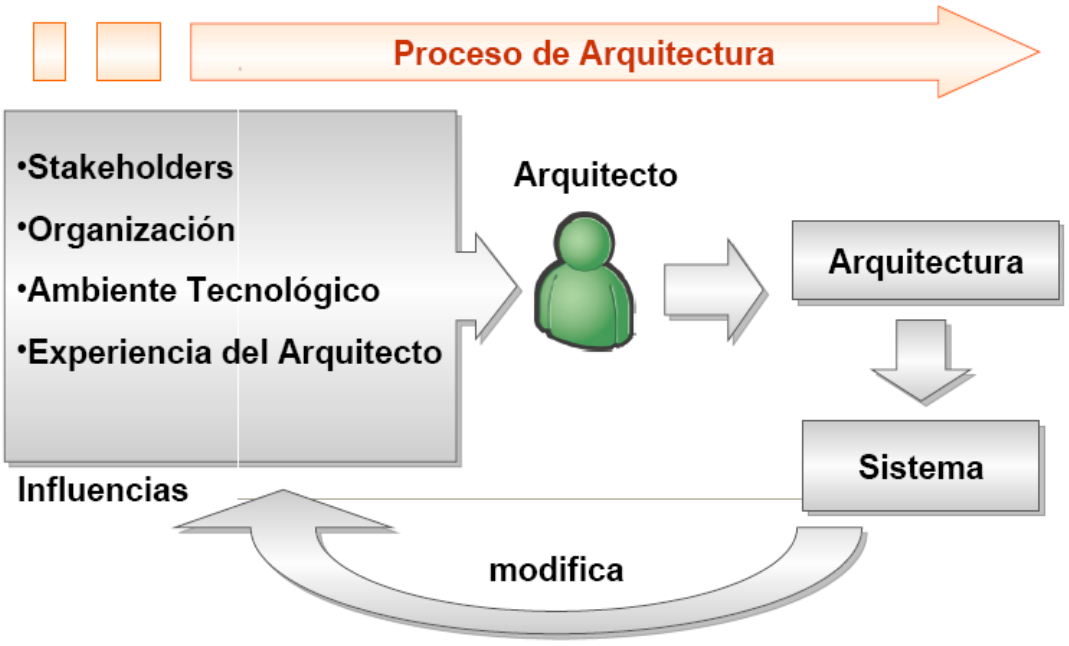

Figura 2. Ciclo de vida de la arquitectura.

Las principales actividades que realiza el arquitecto son:

- Priorizar los Casos de Uso

Definir los Casos de Uso como: críticos, secundarios, auxiliares u opcionales, lo que permite definir los módulos, subsistemas y escenarios así como la interacción entre ellos, que permita tomar decisiones hacia donde centrar los esfuerzos de implementación.

- Análisis de la arquitectura

Definir la arquitectura candidata del sistema teniendo en cuenta arquitecturas similares u otros sistemas, definir además los estilos arquitectónicos, patrones de arquitectura y los principales mecanismos de diseño arquitectónico.

- Identificar mecanismos de diseño

Refinar el análisis de la arquitectura teniendo en cuenta las restricciones impuestas por el entorno de implementación.

- Estructurar el modelo de implementación

Permite establecer la estructura en la que va a residir la implementación del sistema.

- Reutilización de elementos de diseño existentes

Permite analizar las interacciones en los diagramas de clases del Análisis para encontrar interfaces, diseño de clases y subsistemas. Refinar la arquitectura e incorporar elementos reutilizables si es posible, identificar problemas comunes a los que se pueda crear soluciones generales o comunes (patrones, o familias de productos).

- Identificar los elementos de diseño

Analizar las interacciones entre las clases de Análisis para identificar los elementos de modelo de diseño arquitectónico.

- Describir la arquitectura en tiempo de ejecución

Analizar requerimientos de concurrencia, identificar los procesos y la comunicación entre dichos procesos, identificar el ciclo de vida de dichos procesos.

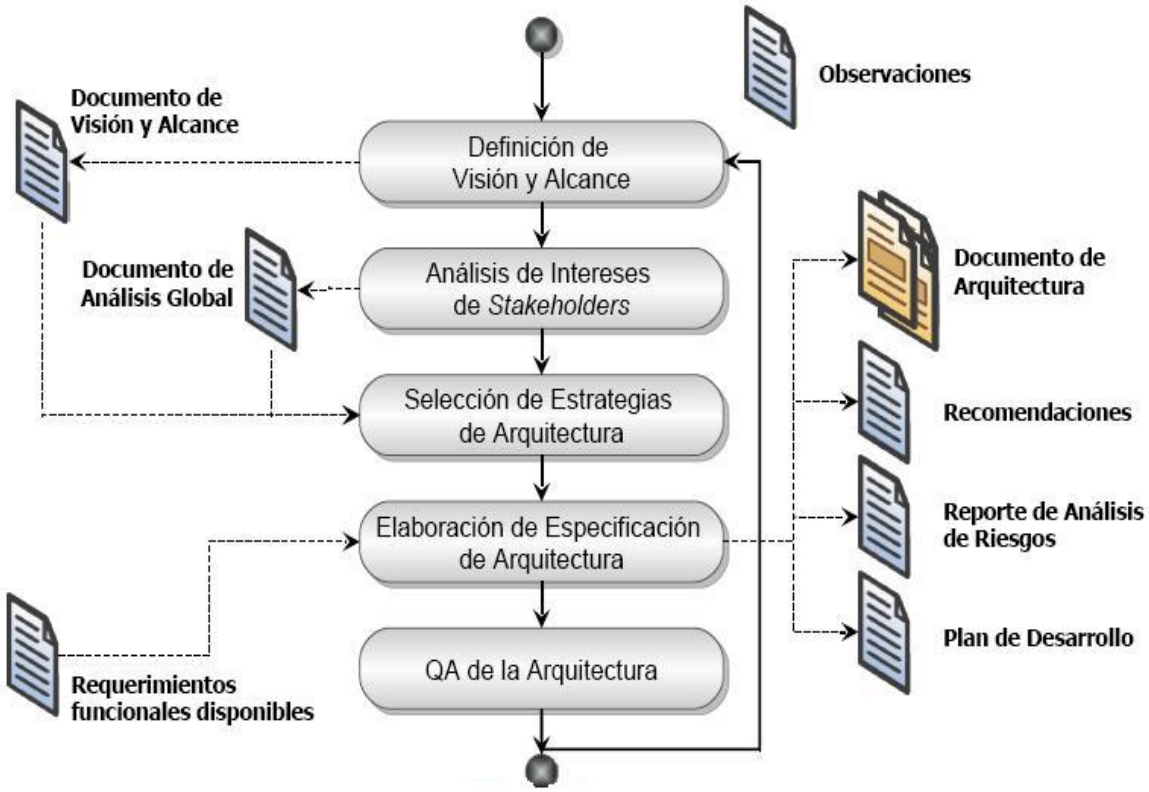

Figura 3. Proceso de Software.

1.5 Lenguajes de Descripción Arquitectónica (ADLs en inglés).

En la década de 1990 y en lo que va del siglo XXI, se han materializado diversas propuestas para describir y razonar en términos de arquitectura de software, muchas de ellas han asumido la forma de lenguajes de descripción de arquitectura, o ADLs, estos ocupan una parte importante del trabajo arquitectónico desde la fundación de la disciplina. Se trata de un conjunto de propuestas de variado nivel de rigurosidad, casi todas ellas de extracción académica, que fueron surgiendo desde comienzos de la década de 1990 hasta la actualidad con el proyecto de unificación de los lenguajes de modelado bajo la forma de UML. Los ADLs difieren sustancialmente de UML, el cual se estima inadecuado en su capacidad

para expresar conectores en particular y en su modelo semántico en general para las clases de descripción y análisis que se requieren. Los ADLs permiten modelar una arquitectura mucho antes que se lleve a cabo la programación de las aplicaciones que la componen, analizar su adecuación, determinar sus puntos críticos y eventualmente simular su comportamiento.

Definición de ADL: 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 (11).

No existe hasta hoy una definición consensuada y unívoca 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.

Criterios de definición de un ADL:

Los ADLs se remontan a los lenguajes de interconexión de módulos (MIL) de la década de 1970, pero se han comenzado a desarrollar con su denominación actual a partir de 1992 o 1993, poco después de fundada la propia arquitectura de software como especialidad profesional. La definición más simple es la de Tracz (12) que define un ADL como una entidad consistente en cuatro “Cs”: componentes, conectores, configuraciones y restricciones (constraints). Una de las definiciones más tempranas es la de Vestal (11) quien sostiene que un ADL debe modelar o soportar los siguientes conceptos:

- Componentes
- Conexiones
- Composición jerárquica, en la que un componente puede contener una sub-arquitectura completa.
- Paradigmas de computación, es decir, semánticas, restricciones y propiedades no funcionales.
- Paradigmas de comunicación.
- Modelos formales subyacentes.
- Soporte de herramientas para modelado, análisis, evaluación y verificación.
- Composición automática de código aplicativo.

Basándose en su experiencia sobre Rapide, Luckham y Vera (13) establecen como requerimientos:

- Abstracción de componentes

- Abstracción de comunicación
- Integridad de comunicación (sólo los componentes que están conectados pueden comunicarse)
- Capacidad de modelar arquitecturas dinámicas
- Composición jerárquica
- Relatividad (o sea, la capacidad de mapear o relacionar conductas entre arquitecturas).

Tomando como parámetro de referencia a UniCon, Shaw y otros (14) alegan que un ADL debe exhibir:

- Capacidad para modelar componentes con aserciones de propiedades, interfaces e implementaciones
- Capacidad de modelar conectores con protocolos, aserción de propiedades e implementaciones
- Abstracción y encapsulamiento
- Tipos y verificación de tipos
- Capacidad para integrar herramientas de análisis.

Después de analizar las opiniones de algunos especialistas del tema se tomará como definición final de ADL la emitida por (15)quien opina que estos deben exhibir:

- **Componentes**
  - Interfaz
  - Tipos
  - Semántica
  - Restricciones (constraints)
  - Evolución
  - Propiedades no funcionales

- **Conectores**
  - Interfaz
  - Tipos
  - Semántica
  - Restricciones
  - Evolución
  - Propiedades no funcionales

- **Configuraciones arquitectónicas**
  - Comprensibilidad
  - Composicionalidad
  - Heterogeneidad
  - Restricciones
  - Refinamiento y trazabilidad.
  - Escalabilidad
  - Evolución
  - Dinamismo
  - Propiedades no funcionales
  - Vistas.

- **Soporte de herramientas**
  - Especificación.
  - Múltiples
  - Análisis.
  - Refinamiento.
  - Generación de código.
  - Dinamismo.

Estos no son más que criterios de evaluación de los ADLs existentes, o sea una especie de referencia para su clasificación y comparación.

1.6 Patrones arquitectónicos.

Buschmann et al. (1996) define *patrón* como una regla formada por 3 partes y representa una relación entre un contexto, un problema y una solución. De forma general un patrón responde a la siguiente estructura:

- Contexto: Es una situación determinada en la cual se muestra un problema determinado.
- Problema: es una proposición que define algo que se desea hacer u obtener y que no se sabe la manera de cómo lograrlo.

- Solución: es la manera en la que se logra hacer algo que se desea.

Los siguientes son algunos de los patrones más usados, divididos en las principales categorías de patrones arquitectónicos (15):

- Lógica de dominio (Domain Logic Patterns)
  - Transaccional (Transaction Script)

    Organiza la lógica del negocio por procedimientos, donde cada procedimiento maneja una sola petición de la presentación. Solo se utiliza en aplicaciones poco complejas.

- Modelo de dominio (Domain Model)

  Modelo del objeto del dominio que incorpora comportamiento y datos.

  - Capa de servicio (Service Layer)

    Define el límite de un uso con una capa de servicios que implante un sistema de operaciones disponibles y coordine la respuesta del uso en cada operación. Este patrón se puede utilizar con el fin de englobar la funcionalidad de la lógica de negocio solo en una capa de la aplicación.

- Patrones Arquitectónicos de Fuente de Datos (Data Source Architectural Patterns)
- Registro Activo (Active Record)

  Un objeto que envuelve una fila en una tabla o una consulta de la base de datos, encapsula el acceso de base de datos, y agrega lógica del dominio en los datos.

- Mapeo de Datos (Data Mapper)

  La capa de Mapeo (Mapper) mueve los datos, convirtiendo los objetos en datos a insertar en las tablas.

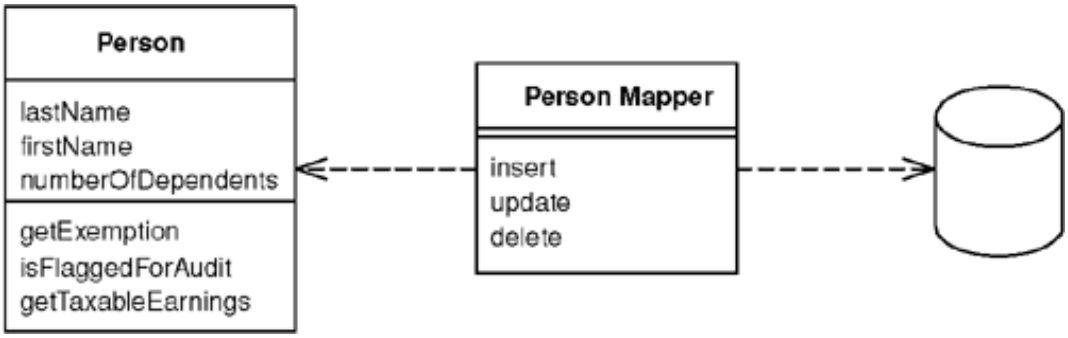

Figura 4. Capa de mapeo (16)

- Comportamiento Objeto-Relacional (Object-Relational Behavioral Patterns)
- Unidad de Trabajo (Unit of Work)

Mantiene una lista de los objetos afectados por una transacción de negocio y coordina poner en escrito los cambios y la resolución de los problemas de la concurrencia.

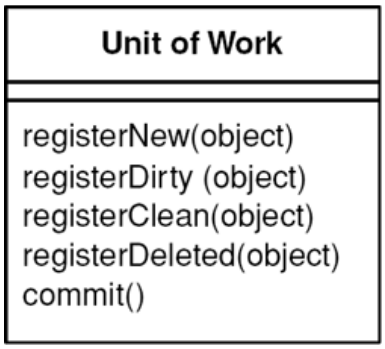

Figura 5. Clase unidad de trabajo (16)

- Mapa de Identificación (Identity Map)

Se asegura de que cada objeto siga cargado solamente una vez manteniendo cada objeto cargado en un mapa.

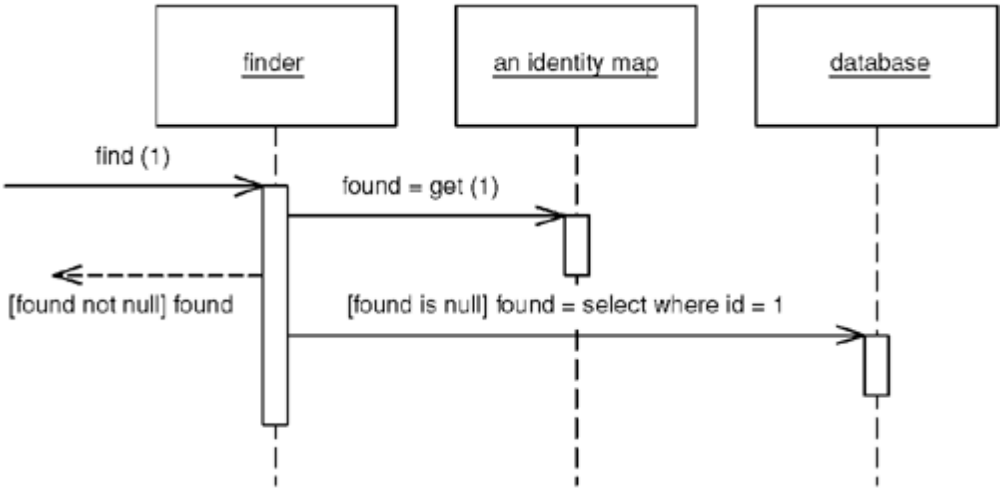

Figura 6. Patrón mapa de identificación (16)

Carga Tardía (Lazy Load)

Un objeto que no contiene todos los datos que se requieren pero sabe conseguirlos.

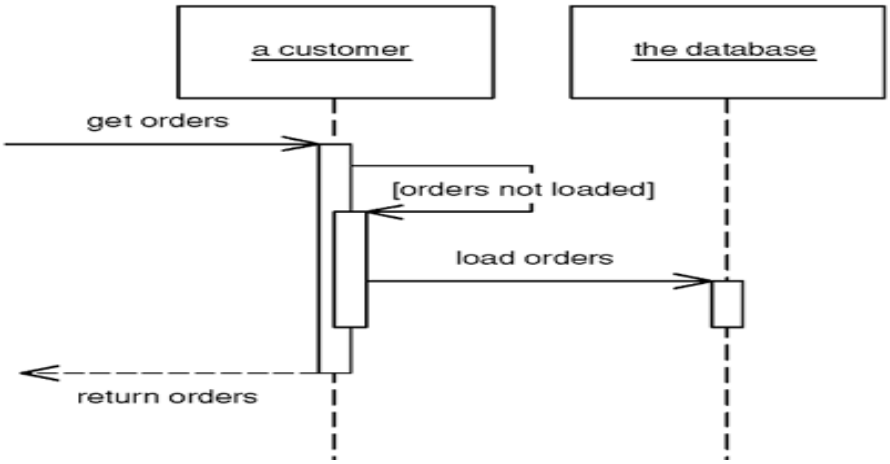

Figura 7. Patrón carga tardía (16)

- Patrones Estructuración Objeto-Relacional (Object-Relational Structural Patterns)
- Mapeo de Llaves Foráneas (Foreign Key Mapping)

Mapea las relaciones en dependencias de las llaves foráneas de las tablas.

- Mapeo de Tablas Asociadas (Association Table Mapping)

  Crea asociaciones entre las tablas asociadas, y los respectivos mapeos de las tablas.

- Mapeo Herencia (Dependent Mapping)

  Hace que una clase realice el acceso a los datos de las clases hijas.

- Patrones de Presentación (Presentation Patterns)
  - Model View Controller (Modelo - Vista - Controlador)

    Divide la interacción con la interfaz de usuario en tres elementos diferentes.

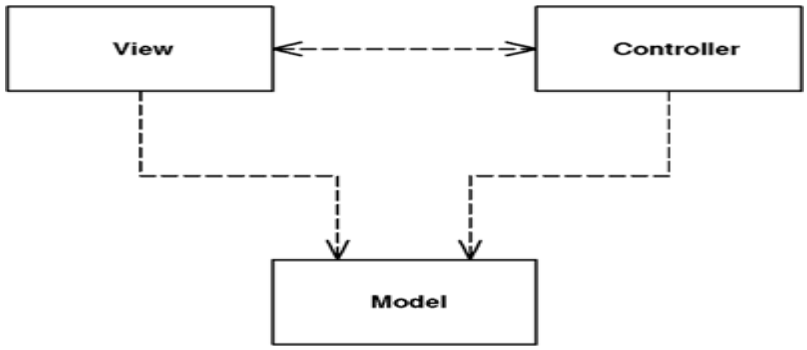

Figura 8. Patrón Modelo – Vista- Controlador (16)

- Patrones de Concurrencia Offline (Offline Concurrency Patterns)
- Bloqueo Offline Pesimista (Pessimistic Offline Lock)

  Previene conflictos entre las transacciones de negocio concurrentes permitiendo que solamente una transacción de negocio a la vez tenga acceso a datos.

- Patrones de Sesión (Session State Patterns)
- Estado de Sesión Cliente (Client Session State)

  Almacena el estado de la sesión en el cliente.

- J2EE Patterns

- Service Locator (Localizador del Servicio)

  Las operaciones de búsqueda y la creación del Servicio implican interfaces y operaciones complejas. Permite no tener que instanciar un servicio.

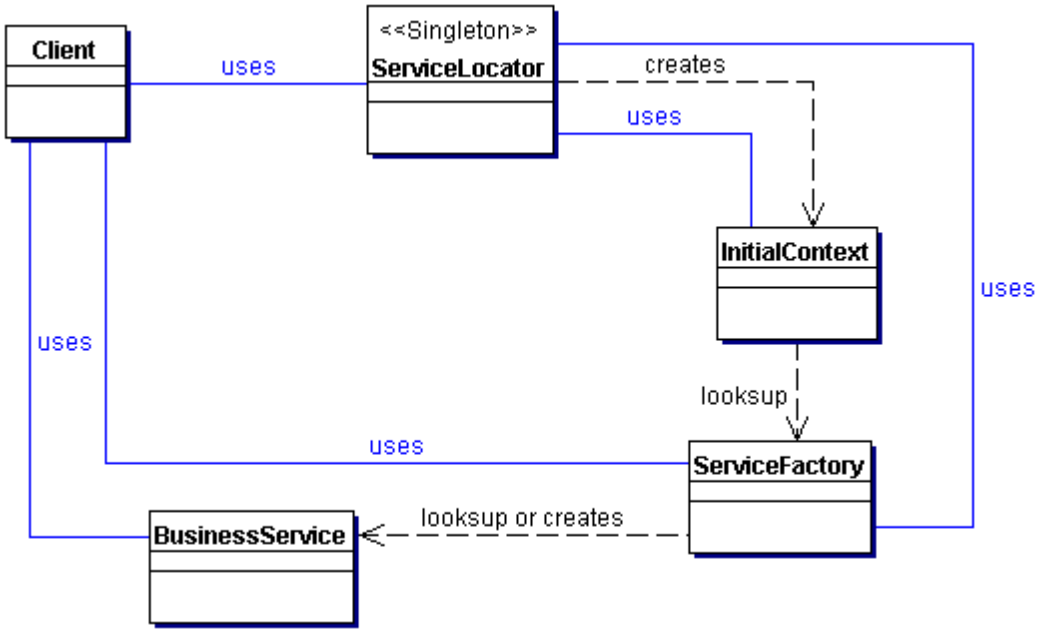

Figura 9. Patrón Service locator (16)

- Objetos de Acceso a Datos (Data Access Object)

  El acceso a los datos varía dependiendo de la fuente de los datos. Tener acceso al almacenamiento persistente, por ejemplo a una base de datos, varía grandemente dependiendo del tipo de almacenamiento.

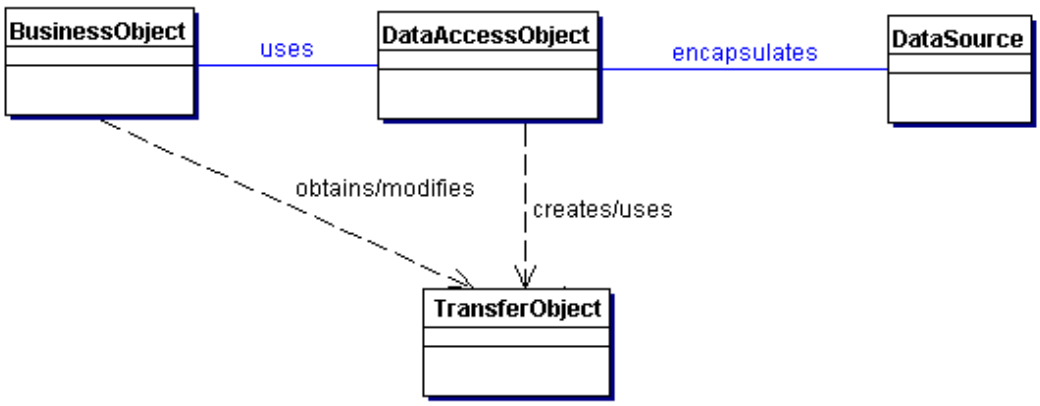

Figura 10. Patrón Objeto de Acceso a Datos (16)

1.7 Estilos Arquitectónicos.

La clave del trabajo arquitectónico tiene que ver con la correcta elección del estilo arquitectónico.

En el caso de los “estilos arquitectónicos” de software son arquitecturas de software comunes, marcos de referencias arquitectónicas, formas comunes o clases de sistemas.

Los estilos de arquitectura se definen como las 4C (17):

- Componentes (Elementos)
- Conectores
- Configuraciones
- Restricciones (Constraints)

A la hora de definir un estilo arquitectónico es necesario tener en cuenta el tipo de aplicación ya que puede imponer restricciones que acotan la interacción de los componentes, además se tiene en cuenta el patrón de organización general. Algunos de los principales estilos que se usan en la actualidad están divididos por Clases de Estilos las que engloban una serie de estilos arquitectónicos específicos:

1.7.1 Estilos de Flujo de datos.

Esta familia de estilos enfatiza la reutilización y la modificabilidad. Es apropiada para sistemas que implementan transformaciones de datos en pasos sucesivos.

- Tuberías y Filtros (estilo arquitectónico específico): Una tubería es una popular arquitectura que conecta componentes computacionales a través de conectores, de modo que el procesamiento de datos se ejecuta como un flujo. Los datos se transportan a través de las tuberías entre los filtros, transformando gradualmente las entradas en salidas. Este estilo se percibe como una serie de transformaciones sobre sucesivas piezas de los datos de entrada.

1.7.2 Estilos centrados en datos.

Esta familia de estilos enfatiza la Integrabilidad de los datos. Se estima apropiada para sistemas que se fundan en acceso y actualización de datos en estructuras de almacenamiento.

- Arquitecturas de Pizarra o repositorio: Esta arquitectura está compuesta por dos componentes principales: una estructura de datos que representa el estado actual y una colección de componentes independientes que operan sobre él. Estos sistemas se han usado en aplicaciones que requieren complejas interpretaciones de proceso de señales (reconocimiento de patrones, reconocimiento de habla).

1.7.3 Estilos de Llamada y Retorno.

Esta familia de estilos enfatiza la modificabilidad y la escalabilidad. Son los estilos más generalizados en sistemas en gran escala.

- Arquitectura en Capas: En este estilo arquitectónico cada capa proporciona servicios a la capa superior y se sirve de las prestaciones que le brinda la inferior, al dividir un sistema en capas, cada capa puede tratarse de forma independiente, sin tener que conocer los detalles de las demás. La división de un sistema en capas facilita el diseño modular, en la que cada capa encapsula un aspecto concreto del sistema y permite además la construcción de sistemas débilmente acoplados, lo que

significa que si se minimiza las dependencias entre capas, resulta más fácil sustituir la implementación de una capa sin afectar al resto del sistema.

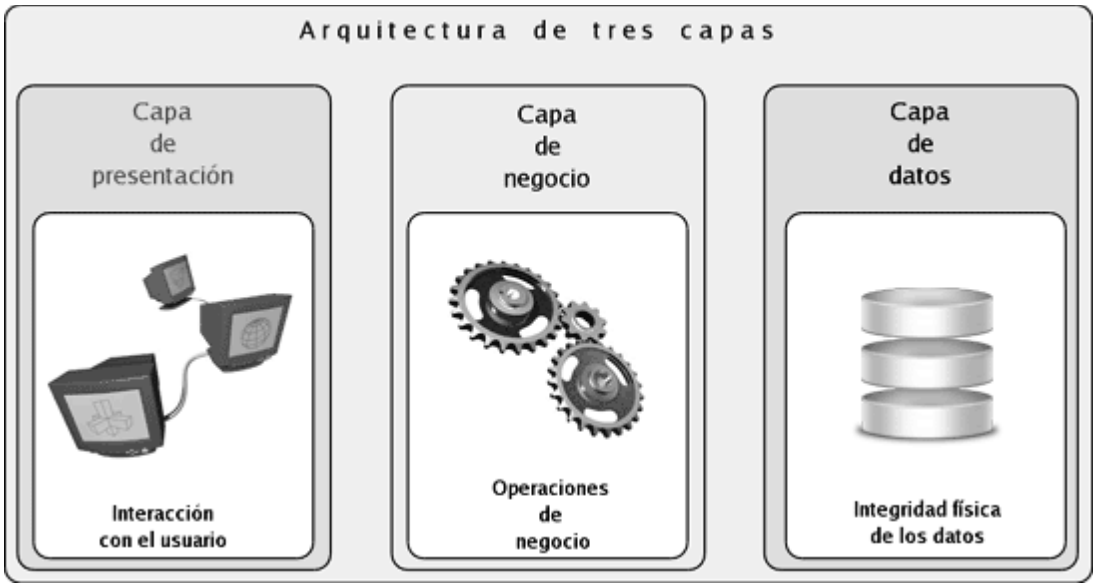

Figura 11. Arquitectura en tres Capas

Las tres capas que propone esta arquitectura son:

- Lógica de Presentación. Esta es la parte del código que interactúa con un dispositivo como una PC o Terminal de autoservicio. Esta capa se encarga de tareas como la disposición de los elementos gráficos en la pantalla, escribir los datos en pantalla, manejo de ventanas, manejo de los eventos del teclado y Mouse, etc.

- Lógica de Negocio. En esta capa se codifican las reglas del negocio. Por ejemplo, si el sistema es de un banco en esta capa se programan conceptos como plazo fijo, cuenta corriente, cheque, etc. Si es una compañía de seguros, existirán componentes para asegurados, pólizas, siniestros, etc.

- Lógica del Procesamiento de los Datos. En esta capa se oculta la forma en que se consultan o almacenan los datos persistentes en la base datos.

Este patrón puede ser difícil a la hora de definir qué componentes ubicar en cada una de las capas, sin embargo mejora el soporte del sistema y facilita la localización de errores.

- Arquitectura Orientada a Objetos: Los componentes de este estilo son los objetos, o más bien instancias de los tipos de dato abstractos. En la caracterización clásica de David Garlan y Mary Shaw (18) los objetos representan una clase de componentes que ellos llaman managers, debido a que son

responsables de preservar la integridad de su propia representación. Un rasgo importante de este aspecto es que la representación interna de un objeto no es accesible desde otros objetos.

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

1.7.4 Estilo de Código Móvil.

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

- Arquitectura de Máquinas Virtuales: Esta arquitectura se conoce como intérpretes basados en tablas ó sistemas basados en reglas. Estos sistemas se representan mediante un seudo-programa a interpretar y una máquina de interpretación. Estas variedades incluyen un extenso espectro que está comprendido desde los llamados lenguajes de alto nivel hasta los paradigmas declarativos no secuenciales de programación.

1.7.5 Estilos Peer To Peer.

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

- Arquitecturas Basadas en Eventos: Estas se han llamado también arquitectura de invocación implícita, estas se vinculan con sistemas basados en publicación-suscripción.

La idea dominante en la invocación implícita es que, en lugar de invocar un procedimiento en forma directa un componente puede anunciar mediante difusión uno o más eventos.

- Arquitecturas Orientadas a Servicios (SOA): Esta construye toda la topología de la aplicación como una topología de interfaces, implementaciones y llamados a interfaces; es una relación entre servicios y consumidores de servicios, ambos lo suficientemente amplios como para representar una función de negocio completa.
- Arquitecturas Basadas en Recursos: Esta define recursos identificables y métodos para acceder y manipular el estado de esos recursos. El caso de referencia es nada menos que la World Wide Web, donde los URLs identifican los recursos y HTTP es el protocolo de acceso. El argumento central es que HTTP mismo, con su conjunto mínimo de métodos y su semántica simplísima, es suficientemente general para modelar cualquier dominio de aplicación.

1.8 Comparación entre Estilo Arquitectónico y Patrón Arquitectónico.

Estilo arquitectónico	**Patrón Arquitectónico**
Describe la estructura general de la aplicación.	Expresa un problema de diseño específico y presenta una solución para él dado el contexto que presenta.
Son independientes del contexto en que puedan ser aplicados.	Existen patrones que definen la estructura básica de una aplicación.
Son una categorización de sistemas.	Requieren de la especificación de un contexto del problema.
Expresan técnicas de diseño desde una perspectiva independiente a la situación del diseño.	Son soluciones generales a problemas comunes.
Expresa componentes y relaciones.	Es una plantilla de construcción.

Tabla 1 comparativa de Estilos y Patrones Arquitectónicos

1.9 Calidad de la arquitectura.

Barbacci y otros establecen que el desarrollo de formas sistemáticas para relacionar atributos de calidad de un sistema a su arquitectura provee una base para la toma de decisiones objetivas sobre acuerdos de diseño y permite a los ingenieros realizar predicciones razonablemente exactas sobre los atributos del sistema que son libres de prejuicios y asunciones no triviales. El objetivo de fondo es la habilidad de evaluar cuantitativamente y llegar a acuerdos entre múltiples atributos de calidad para alcanzar un mejor sistema de forma global. (19)

1.10 Atributos de calidad.

Se conoce como atributos de calidad a aquellas características que definen propiedades de un servicio que presta el sistema a sus usuarios.

Bass et al. (1998) establece una clasificación de los atributos de calidad en dos categorías:

- Observables vía ejecución: aquellos atributos que se determinan del comportamiento del sistema en tiempo de ejecución. La descripción de algunos de estos atributos se presenta en la Tabla 2.
- No observables vía ejecución: aquellos atributos que se establecen durante el desarrollo del sistema.

La descripción de algunos de estos atributos se presenta en la Tabla 3.

Atributo de calidad	**Descripción**
Disponibilidad (*Availability*)	Es la medida de disponibilidad del sistema para el uso. (19)
Confidencialidad *(Confidentiality)*	Es la ausencia de acceso no autorizado a la información. (19)

Funcionalidad *(Functionality)*	Habilidad del sistema para realizar el trabajo para el cual fue concebido. (20)
Desempeño *(Performance)*	El desempeño de un sistema se refiere a aspectos temporales del comportamiento del mismo, a la capacidad de respuesta, ya sea el tiempo requerido para responder a aspectos específicos o el número de eventos procesados en un intervalo de tiempo.
Confiabilidad *(Reliability)*	Es la medida de la habilidad de un sistema a mantenerse operativo a lo largo del tiempo. (19)
Seguridad externa *(Safety)*	Ausencia de consecuencias catastróficas en el ambiente. Es la medida de ausencia de errores que generan pérdidas de información. (19)

Tabla 2 Descripción de atributos de calidad observables vía ejecución

Atributo de calidad	**Descripción**
Configurabilidad *(Configurability)*	Posibilidad que se otorga a un usuario experto a realizar ciertos cambios al Sistema. (21)
Integrabilidad *(Integrability)*	Es la medida en que trabajan correctamente los componentes del sistema que fueron desarrollados separadamente al ser integrados. (22)
Integridad *(Integrity)*	Es la ausencia de alteraciones inapropiadas de la información. (19)
Interoperabilidad	Es la medida de la habilidad de que un grupo de partes del sistema trabajen

(Interoperability)	con otro sistema. Es un tipo especial de Integrabilidad (22)
Modificabilidad *(Modifiability)*	Es la habilidad de realizar cambios futuros al sistema. (21)
Mantenibilidad *(Maintainability)*	Es la capacidad de someter a un sistema a reparaciones y evolución. (19)
Portabilidad *(Portability)*	Es la habilidad del sistema para ser ejecutado en diferentes ambientes de computación. Estos ambientes pueden ser hardware, software o una combinación de los dos. (20)
Reusabilidad *(Reusability)*	Es la capacidad de diseñar un sistema de forma tal que su estructura o parte de sus componentes puedan ser reutilizados en futuras aplicaciones. (21)
Escalabilidad *(Scalability)*	Es el grado con el que se pueden ampliar el diseño arquitectónico, de datos o procedimental. (23)
Capacidad de Prueba *(Testability)*	Es la medida de la facilidad con la que el software, al ser sometido a una serie de pruebas, puede demostrar sus fallas. Es la probabilidad de que, asumiendo que tiene al menos una falla, el software fallará en su próxima ejecución de prueba. (22)

Tabla 3 Descripción de atributos de calidad no observables vía ejecución

En su mayoría, los atributos de calidad se pueden organizar y descomponer de maneras diferentes, en lo que se conoce como *modelos de calidad*. Los modelos de calidad de software facilitan el entendimiento del proceso de la Ingeniería de Software. (23)

1.11 Conclusiones.

Después de analizar la arquitectura de software a lo largo de este capítulo, queda evidenciada la importancia de esta disciplina en el proceso de desarrollo de software, ya que esta contempla el aseguramiento de los requisitos, tanto funcionales como no funcionales que debe cumplir el sistema para su óptimo comportamiento. Además su descripción es fundamental e imprescindible pues en esta etapa se obtiene una propuesta de solución que debe resolver lo más eficiente posible los problemas del cliente.

Capítulo 2: Análisis de algunos ADLs y propuesta para utilizar en la aplicación del caso de estudio.

2.1 Introducción.

En el presente capítulo se llevará a cabo una comparación entre UML y los ADL, basándose en varios criterios que resultan importantes en la Descripción de la Arquitectura, así como un análisis de los principales ADLs que se utilizan en la actualidad, atendiendo a sus características esenciales y dinamismo.

2.2 Comparación entre el Lenguaje Unificado de Modelado (UML) y Lenguajes de descripción de arquitectura (ADL).

Antes de realizar un detallado análisis de algunos de los Lenguajes de Descripción de Arquitectura más utilizados en la actualidad se pretende realizar una comparación entre El Lenguaje Unificado de Modelado (UML) y los ADLs de manera general, lo que conduce a realizarse la siguiente interrogante: ¿Qué es recomendable utilizar para la descripción de la Arquitectura de Software de una forma más eficiente?

Lenguaje Unificado de Modelado (UML)

El Lenguaje Unificado de Modelado (UML) es una notación para especificar, visualizar y documentar sistemas de software desde la perspectiva de la orientación a objetos. Su objetivo es representar el conocimiento acerca de los sistemas que se pretenden construir y las decisiones tomadas durante su desarrollo, tanto lo referido a su estructura estática como a su comportamiento dinámico. UML postula un proceso de desarrollo iterativo, incremental, guiado por los casos de uso y *centrado en la arquitectura* (21). La representación en UML de un sistema de software consta de cinco vistas o modelos parciales separados, aunque relacionados entre sí, denominadas vista de casos de uso, de diseño, de implementación, de procesos y de despliegue. Cada uno de estos modelos representa el sistema por

medio de diversos diagramas. Aunque no existe de forma explícita una vista arquitectónica, estas cinco vistas pretenden describir, en su conjunto, la arquitectura del sistema (24).

En favor de UML se puede señalar que es un lenguaje gráfico con sintaxis y semántica bastante bien definidas. La sintaxis de la notación gráfica se especifica mediante su correspondencia con los elementos del modelo semántico subyacente (21), cuya semántica se define de manera semi-formal por medio de un metamodelo, textos descriptivos y restricciones. Existen además numerosas iniciativas para dotar a esta notación de una semántica formal (25) (26) , (27) (28).

Por otra parte, el lenguaje es extensible, de forma que pueden añadirse nuevas construcciones para abordar aspectos del desarrollo de software no previstos inicialmente en la notación. Esta extensión puede realizarse bien mediante la especialización de los conceptos del metamodelo (lo que hace que la notación extendida ya no sea compatible con las herramientas existentes), o bien mediante la definición de restricciones, valores etiquetados y *estereotipos*, sin modificar la sintaxis ni la semántica de UML (28).

En cuanto a su capacidad para describir la arquitectura, se debe señalar que UML maneja conceptos utilizados también por la Arquitectura de Software, como son: interfaz, componente o conexión. Además los mecanismos de extensión disponibles pueden utilizarse para definir otros conceptos no contemplados o para establecer restricciones que definan de forma más precisa la semántica de estos conceptos.

No obstante, partiendo de la base de que el propósito primordial de un lenguaje es proporcionar un vehículo de expresión para las nociones intuitivas y prácticas de sus usuarios (29), en este caso, los arquitectos del software. Si ciertas abstracciones fundamentales utilizadas por los arquitectos (por ejemplo, los componentes o los conectores) quedan ocultas o desvirtuadas debido a su representación mediante las abstracciones que proporciona el lenguaje (en este caso las clases), la labor del arquitecto se ve dificultada de forma innecesaria (30). Se concluye, por tanto, que a pesar de las afirmaciones de sus autores, el uso previsto de UML no es la descripción de la arquitectura del software como tal, de forma que la expresividad para modelar sistemas ofrecida por UML no satisface de manera completa las necesidades de la descripción arquitectónica. En particular, resulta deficiente para la descripción de los aspectos dinámicos de las estructuras, los protocolos de comunicación entre componentes y la correspondencia entre las distintas vistas de la arquitectura (31).

En términos de número, los conocedores de UML no admiten comparación con la todavía modesta población de especialistas en ADLs. En la comunidad de arquitectos existen dos líneas claramente definidas; la primera, vinculada con el mundo de Rational y UML, la cual impulsa el uso casi absoluto de UML como si fuera un ADL normal y la segunda ha señalado reiteradas veces las limitaciones de UML no sólo como ADL sino como lenguaje universal de modelado. La literatura crítica de UML es ya un tópico clásico de la reciente arquitectura de software. Entre las deficiencias que se han señalado de UML como lenguaje de especificación están las siguientes:

1.	Un caso de uso de UML no puede especificar los requerimientos de interacción en situaciones en las que un sistema deba iniciar una interacción entre él mismo y un actor externo, puesto que proscribe asociaciones entre actores (32).
2.	Al menos en algunas especificaciones, como UML 1.3, es imposible expresar relaciones secuenciales, paralelas o iterativas entre casos de uso, salvo mediante extensiones o estereotipos que introducen oscuridad en el diseño del sistema; además, prohíbe la descomposición de casos de uso y la comunicación entre ellos.
3.	UML tampoco puede expresar una estructura entre casos de uso ni una jerarquía de casos de una forma fácil y directa; una precondición tal como “el caso de uso A requiere que el caso de uso X haya sido ejecutado previamente” no se puede expresar formalmente en ese lenguaje.
4.	Un modelo de casos de uso tampoco puede expresar adecuadamente la conducta de un sistema dependiente de estados, o la descomposición de un sub-sistema distribuido. El modelado del flujo de información en un sistema consistente en subsistemas es al menos confuso en UML, y no se puede expresar en una sola vista (32).
5.	Otros autores, como Theodor Tempelmeier (33) han opuesto objeciones a UML como herramienta de diseño en el caso de sistemas embebidos, de alta concurrencia, de misión crítica o tiempo real, así como han señalado su inutilidad casi total para modelar sistemas con cualidades emergentes o sistemas que involucren representación del conocimiento.
6.	Otros autores dedicaron tesis enteras a las dificultades de UML referidas a componentes reflexivos y aspectos. Mientras que el modelado orientado a objetos especifica con claridad la interfaz de inbound de un objeto (el lado “tiene” de una interfaz), casi ninguna herramienta permite expresar con naturalidad la interfaz opuesta de outbound, popularmente conocida como “necesita”. Sólo el diseño por contrato de Bertrand Meyer soporta este método de dos vías en su núcleo; en UML se debió implementar como extensión o metamodelo. UML tampoco considera los conectores como objetos de primera clase, por lo cual se deben implementar extensiones mediante Lenguaje de Modelado de Objetos en Tiempo Real (ROOM) o de alguna otra manera (34).

7.	Klaus-Dieter Schewe, en particular, ha reseñado lo que él interpreta como un cúmulo de limitaciones y oscuridades de UML, popularizando además su consideración como un “dinosaurio moderno” (47). Al cabo de un análisis pormenorizado del que aquí no se puede dar detalle sin excederse de espacio, Schewe estima que, dando continuidad a una vieja polémica iniciada por E. F. Codd cuando éste cuestionó a los modelos en Entidad-Relación, UML es el ganador contemporáneo cuando se trata de falta de definiciones precisas, falta de una clara semántica, falta de claridad con respecto a los niveles de abstracción y falta de una metodología pragmática.
8.	(35) Alegan que UML es deficiente para describir correspondencias tales como el mapeo entre elementos en diferentes vistas, que serían mejor representadas en una tabla; faltan también en UML los elementos para modelar comunicación peer-to-peer. Los diagramas de secuencia de UML no son tampoco adecuados para soportar la configuración dinámica de un sistema, ni para describir secuencias generales de actividades.

Tabla 4. Descripción de atributos de calidad no observables vía ejecución

Lenguajes de descripción de arquitectura (ADL).

Según lo anteriormente expuesto, resulta evidente la necesidad de una notación específicamente desarrollada para la descripción de la arquitectura del software, de forma que permita abordar de manera adecuada los problemas y necesidades de este nivel de diseño. Estos son los denominados lenguajes de descripción de arquitectura. Este tipo de lenguajes son necesarios para disponer de abstracciones útiles para modelar sistemas complejos desde un punto de vista arquitectónico, a la vez que deben permitir un nivel de detalle suficiente como para describir propiedades de interés de dichos sistemas. De esta manera será posible comprobar, ya desde las etapas iniciales del desarrollo de un sistema, si este cumple o no determinados requisitos.

No existe aún un consenso claro dentro de la comunidad científica respecto a lo que es un ADL, ni sobre qué aspectos de los sistemas de software deben ser contemplados por este tipo de lenguajes. No obstante, entre las características de estos lenguajes se pueden considerar las siguientes (10):

- Composición: Permiten la representación del sistema como composición de una serie de partes.
- Configuración: La descripción de la arquitectura es independiente de la de los componentes que formen parte del sistema.

- Abstracción: Describen los *roles* o papeles abstractos que juegan los componentes dentro de la arquitectura.
- Flexibilidad: Permiten la definición de nuevas formas de interacción entre componentes.
- Reutilización: Permiten la reutilización tanto de los componentes como de la propia arquitectura.
- Heterogeneidad: Permiten combinar descripciones heterogéneas.
- Análisis: Permiten diversas formas de análisis de la arquitectura y de los sistemas desarrollados a partir de ella.

De un modo general, se podría decir que un ADL se centra en la estructura de alto nivel de un sistema de software, más que en los detalles de implementación de los módulos de código fuente que lo constituyen. Los ADLs proporcionan tanto una sintaxis específica como un marco conceptual para modelar la arquitectura de un sistema de software (11). Los conceptos fundamentales manejados en una descripción arquitectónica son los siguientes:

- Componentes. Representan unidades de computación o de almacenamiento de datos.
- Conectores. Son utilizados para modelar las interacciones entre componentes y las reglas que gobiernan dichas interacciones.
- Configuraciones arquitectónicas. Grafos de componentes y conectores que describen la estructura arquitectónica de los sistemas de software.

El objeto no es describir los detalles internos de los componentes del sistema, sino sólo su interfaz, es decir la información necesaria para interconectar dichos componentes y formar así sistemas mayores. Como se verá más adelante, esta descripción de las interfaces no se limita —tal como sucede en los lenguajes de programación convencionales— a la signatura de los métodos que ofrece o requiere el componente, sino que incluye información sobre la funcionalidad del componente, los patrones de interacción que utiliza en su funcionamiento, y otras características diversas. Si además de esto se pretende inferir algún tipo de propiedad a partir de una descripción arquitectónica, las interfaces deben estar modeladas formalmente.

De acuerdo con esto, el marco conceptual de un ADL suele incluir una teoría formal, en términos de la cual se expresa la semántica del lenguaje. El marco formal elegido determina la adecuación del ADL para modelar determinados tipos de sistemas (por ejemplo, sistemas altamente concurrentes, sistemas

organizados en capas, etc.) o para modelar determinados aspectos de un sistema (por ejemplo, propiedades de viveza, propiedades de tiempo real, etc.)

A pesar de la relativa juventud de este campo, existen diversos ejemplos de ADLs. En estos lenguajes se separa claramente lo que es la descripción de la arquitectura o configuración del sistema de la descripción e implementación de los componentes. Además, muchos de ellos tienen una base formal (por lo general un álgebra de procesos), lo que permite el análisis y verificación de propiedades de los sistemas descritos. No obstante, el mayor inconveniente que presentan es que se está tratando con un campo relativamente reciente y por tanto muy inmaduro e inestable, es por ello que la difusión de estos lenguajes es aún reducida.

En realidad, la mayoría de los ADLs existentes carecen de alguna de estas propiedades, pero tomada en un sentido amplio esta caracterización resulta suficiente para delimitar con claridad qué es lo que no constituye un ADL propiamente dicho.

A continuación se analizarán algunos de los principales ADLs atendiendo a su disponibilidad para la plataforma Windows, las herramientas gráficas que posee, así como su capacidad de de generar código ejecutable, principales variables a medir en los lenguaje de especificación de arquitectura.

2.3 Tipos de ADL.

2.3.1 Acme

Acme se define como una herramienta capaz de soportar el mapeo de especificaciones arquitectónicas entre diferentes ADLs o sea que es un lenguaje de intercambio de arquitectura, por lo que no se considera un ADL propiamente dicho aunque se trata como tal pues posee prestaciones propias de los lenguajes de especificación de arquitectura. Se reconoce que Acme no es apto para cualquier tipo de sistemas, destacando su usabilidad en sistemas relativamente simples.

El proyecto Acme comenzó a principios de 1995 en la Escuela de Ciencias de la Computación de la Universidad Carnegie Mellon. Hoy este proyecto se organiza en dos grandes grupos, que son el lenguaje Acme propiamente dicho y el Acme Tool Developer's Library (AcmeLib). David Garlan ha sido uno de los

más destacados arquitectos que ha trabajado en el desarrollo de esta herramienta, uno de los teóricos de arquitectura de software más activo en la década de los 90.

Objetivo principal – La motivación fundamental de Acme es el intercambio entre arquitecturas e integración de ADLs. Garlan considera que Acme es un lenguaje de descripción arquitectónica de segunda generación; podría decirse que es de segundo orden: un metalenguaje, una lengua franca para el entendimiento de dos o más ADLs, incluido Acme mismo. Con el tiempo, sin embargo, la dimensión metalingüística de Acme fue perdiendo prioridad y los desarrollos actuales profundizan su capacidad intrínseca como ADL puro.

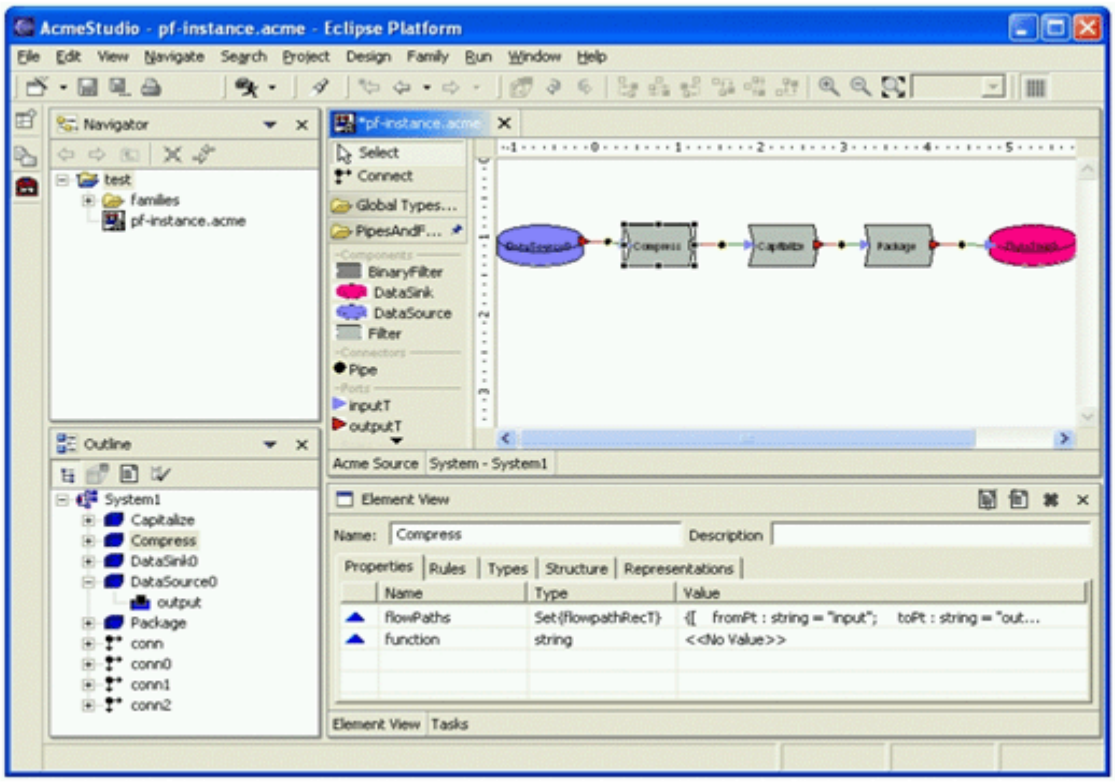

Figura 12. Ambiente de edición de AcmeStudio con diagrama de tubería y filtros.

Acme soporta la definición de cuatro tipos de arquitectura:

Interfaces: Todos los ADLs conocidos soportan la especificación de interfaces para sus componentes. En Acme cada componente puede tener múltiples interfaces. Los puertos pueden definir interfaces tanto simples como complejas, desde una signatura de procedimiento hasta una colección de rutinas a ser invocadas en cierto orden, o un evento de multicast.

Conectores: En su ejemplar estudio de los ADLs existentes (36), llama a los lenguajes que modelan sus conectores como entidades de lenguajes de configuración explícitos, en oposición a los lenguajes de configuración in-line. Acme pertenece a la primera clase. Los conectores representan interacciones entre componentes, también tienen interfaces que están definidas por un conjunto de roles. Además existen conectores binarios que son los más sencillos: el invocador y el invocado de un conector RPC, la lectura y la escritura de un conector de tubería, el remitente y el receptor de un conector de paso de mensajes.

Semántica: Muchos lenguajes de tipo ADL no modelan la semántica de los componentes más allá de sus interfaces. En este sentido, Acme sólo soporta cierta clase de información semántica en listas de propiedades. Estas propiedades no se interpretan, y sólo existen a efectos de documentación.

Estilos: Acme posee manejo intensivo de familias o estilos. Esta capacidad está construida naturalmente como una jerarquía de propiedades correspondientes a tipos. Acme considera, en efecto, tres clase de tipos: tipos de propiedades, tipos estructurales y estilos. Así como los tipos estructurales representan conjuntos de elementos estructurales, una familia o estilo representa un conjunto de sistemas. Una familia Acme se define especificando tres elementos de juicio: un conjunto de tipos de propiedades y tipos estructurales, un conjunto de restricciones y una estructura por defecto, que prescribe el conjunto mínimo de instancias que debe aparecer en cualquier sistema de la familia.

Además en ACME se pueden crear sistemas (system), los cuales son un conjunto de componentes y conectores describiendo la interacción entre ellos. Estos sistemas pueden considerarse entidades de ACME pues a través de sus propiedades pueden describirse atributos del sistema del cual se está describiendo la arquitectura.

La biblioteca AcmeLib define un conjunto de clases para manipular representaciones arquitectónicas Acme en cualquier aplicación. Su código se encuentra disponible tanto en C++ como en Java, y puede ser invocada por lo tanto desde cualquier lenguaje la plataforma clásica de Microsoft o desde el framework de

.NET. Es posible entonces implementar funcionalidad conforme a Acme en forma directa en cualquier programa, o llegado el caso (mediante wrapping) exponer Acme como web service.

2.3.2 Jacal.

Es un lenguaje de descripción de arquitecturas de software de propósito general creado en la Universidad de Buenos Aires, por un grupo de investigación del Departamento de Computación de la Facultad de Ciencias Exactas y Naturales.

Objetivo principal – El objetivo principal de Jacal es lo que actualmente se denomina “animación” de arquitecturas. Esto es, poder visualizar una simulación de cómo se comportaría en la práctica un sistema basado en la arquitectura que se ha representado.

Más allá de este objetivo principal, el diseño de Jacal contempla otras características deseables en un ADL, como por ejemplo contar con una representación gráfica que permita a simple vista transmitir la arquitectura del sistema, sin necesidad de recurrir a información adicional. Para este fin, se cuenta con un conjunto predefinido (extensible) de conectores, cada uno con una representación distinta.

Estilos: Jacal no cuenta con una notación particular para expresar estilos, aunque por tratarse de un lenguaje de propósito general, puede ser utilizado para expresar arquitecturas de distintos estilos. No ofrece una forma de restringir una configuración a un estilo específico, ni de validar la conformidad.

Interfaces: Cada componente cuenta con puertos (ports) que constituyen su interfaz y a los que pueden adosarse conectores.

Semántica: Jacal tiene una semántica formal que está dada en función de redes de Petri. Se trata de una semántica denotacional que asocia a cada arquitectura una red correspondiente. La semántica operacional estándar de las redes de Petri es la que justifica la animación de las arquitecturas.

Interfaz gráfica: Como ya se ha dicho, la notación principal de Jacal es gráfica y hay una herramienta disponible en línea para editar y animar visualmente las arquitecturas.

Análisis y verificación: Las animaciones de arquitecturas funcionan como casos de prueba. La herramienta de edición y animación disponible en el sitio del proyecto permite dibujar arquitecturas

mediante un editor orientado a la sintaxis, para luego animarlas y almacenar el resultado de las ejecuciones en archivos de texto. Esta actividad se trata exclusivamente de una tarea de testing, debiendo probarse cada uno de los casos que se consideren críticos, para luego extraer conclusiones del comportamiento observado o de las trazas generadas. Si bien no se ofrecen actualmente herramientas para realizar procesos de verificación automática como modelchecking, la traducción a redes de Petri ofrece la posibilidad de aplicar al resultado otras herramientas disponibles en el mercado.

Generación de código: En su versión actual, Jacal no genera código de ningún lenguaje de programación, ya que no fuerza ninguna implementación única para los conectores.

Disponibilidad de plataforma: La herramienta que actualmente está disponible para editar y animar arquitecturas en Jacal es una aplicación Win32, que no requiere instalación, basta con copiar el archivo ejecutable para comenzar a usarla.

El ambiente consiste en una interfaz gráfica de usuario, donde pueden dibujarse representaciones Jacal de sistemas, incluyendo tanto el nivel de interfaz como el de comportamiento. Se pueden editar múltiples sistemas simultáneamente y, abriendo distintas vistas, visualizar simultáneamente los dos niveles de un mismo sistema, para uno o más componentes. El editor es orientado a la sintaxis, en el sentido de que no permite dibujar configuraciones inválidas. Por ejemplo, valida la compatibilidad entre el tipo de un componente y los conectores asociados. Para aumentar la flexibilidad (especialmente en el orden en que se dibuja una arquitectura), se dejan otras validaciones para el momento de la animación. (Aparece alguna bibliografía referenciada pero dice que no esta publicada)

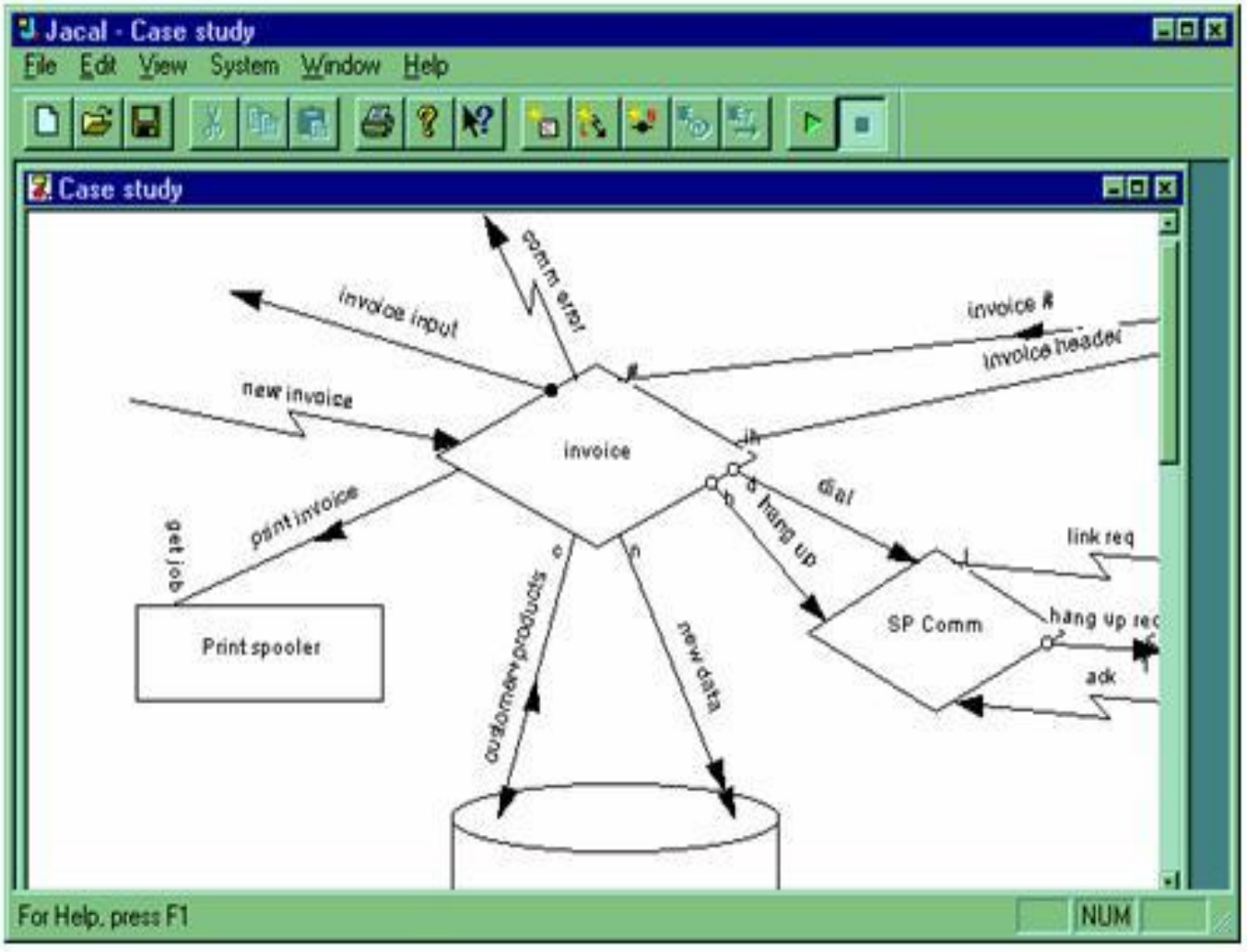

Figura 13. Representación grafica de una arquitectura en Jacal

2.3.3 UniCon.

UniCon (Universal Connector Support) es un ADL desarrollado por Mary Shaw y otros (29). Proporciona una herramienta de diseño para construir configuraciones ejecutables basadas en tipos de componentes, implementaciones y “conexiones expertas” que soportan tipos particulares de conectores. UniCon se asemeja a Darwin en la medida en que proporciona herramientas para desarrollar configuraciones

ejecutables de caja negra y posee un número fijo de tipos de interacción, pero el modelo de conectores de ambos ADLs es distinto.

Oficialmente se define como un ADL cuyo foco apunta a soportar la variedad de partes y estilos que se encuentra en la vida real y en la construcción de sistemas a partir de sus descripciones arquitectónicas. UniCon es el ADL propio del proyecto Vitruvius, cuyo objetivo es elucidar un nivel de abstracción de modo tal que se pueda capturar, organizar y tornar disponible la experiencia colectiva exitosa de los arquitectos de software.

Objetivo principal: El propósito de UniCon es generar código ejecutable a partir de una descripción, a partir de componentes primitivos adecuados. UniCon se destaca por su capacidad de manejo de métodos de análisis de tiempo real a través de RMA (Rate Monotonic Analysis).

Estilos: UniCon no proporciona medios para describir o delinear familias de sistemas o estilos.

Interfaces: En UniCon los puntos de interfaces de los componentes se llaman players. Estos players poseen un tipo que indica la naturaleza de la interacción esperada, y un conjunto de propiedades que detalla la interacción del componente en relación con esa interfaz. En el momento de configuración, los players de los componentes se asocian con los roles de los conectores.

Semántica: UniCon sólo soporta cierta clase de información semántica en listas de propiedades.

2.3.4 Aesop.

El nombre oficial es Aesop (Software Architecture Design Environment Generator). Se ha desarrollado como parte del proyecto ABLE de la Universidad Carnegie Mellon, cuyo objetivo es la exploración de las bases formales de la arquitectura de software, el desarrollo del concepto de estilo arquitectónico y la producción de herramientas útiles a la arquitectura, de las cuales Aesop es precisamente la más relevante. La definición también oficial de Aesop es “una herramienta para construir ambientes de diseño de software basada en principios de arquitectura”. El ambiente de desarrollo de Aesop System se basa en el estilo de tubería y filtros propio de UNIX. Un diseño en Aesop requiere manejar toda una jerarquía de lenguajes específicos, y en particular FAM Command Language (FCL) que a su vez es una extensión de Lenguaje de Herramientas de Comandos (TCL) orientada a soportar modelado arquitectónico. FCL es una

combinación de TCL y C densamente orientada a objetos. En lo que respecta al manejo de métodos de análisis de tiempo real, Aesop implementa EDF (Earliest Deadline First).

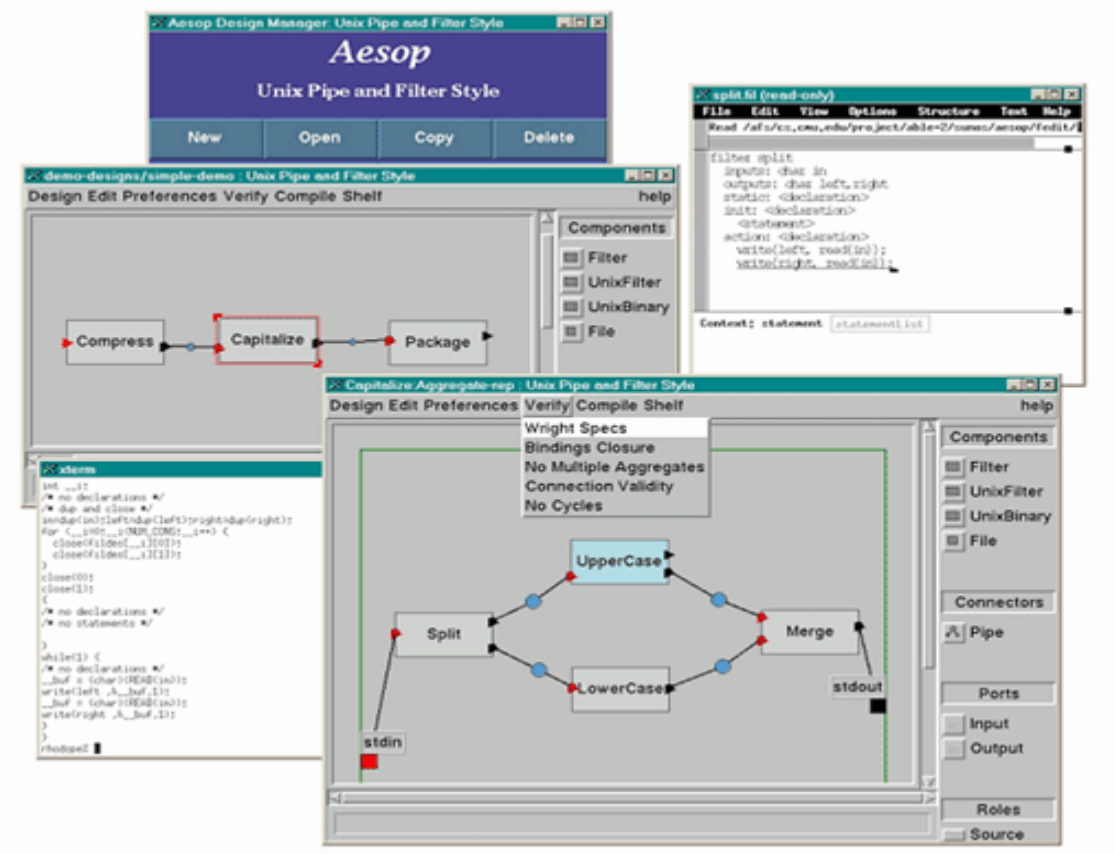

Figura 14 Ambiente gráfico de Aesop con diagrama de tubería y filtro.

Estilos: En Aesop, conforme a su naturaleza orientada a objetos, el vocabulario relativo a estilos arquitectónicos se describe mediante la definición de sub-tipos de los tipos arquitectónicos básicos: Componente, Conector, Puerto, Rol, Configuración y Binding.

Interfaces: En Aesop los puntos de interfaz se llaman puertos (ports).

Modelo semántico: Aesop presupone que la semántica de una arquitectura puede ser arbitrariamente distinta para cada estilo. Por lo tanto, no incluye ningún soporte nativo para la descripción de la semántica

de un estilo o configuración, sino que apenas presenta unos cuadros vacantes para colocar esa información como comentario.

Soporte de lenguajes: Aesop (igual que Darwin) sólo soporta nativamente desarrollos realizados en C++.

Generación de código: Aesop genera código C++. Aunque Aesop opera primariamente desde una interfaz visual, el código de Aesop es marcadamente más procedural que el de Acme, por ejemplo, el cual posee una estructura de orden más bien declarativo.

Disponibilidad de plataforma: Aesop no está disponible en plataforma Windows, aunque naturalmente puede utilizarse para modelar sistemas implementados en cualquier plataforma.

Aunque hay bastante información en línea, el vínculo de distribución de Aesop estaba muerto o inaccesible en el momento de redacción de este documento.

2.3.5 Darwin.

Darwin es un lenguaje de descripción arquitectónica desarrollado por Jeff Magee y Jeff Kramer. Darwin describe un tipo de componente mediante una interfaz consistente en una colección de servicios que son: provistos (declarados por ese componente) o requeridos (o sea, que se espera ocurran en el entorno). Las configuraciones se desarrollan instanciando las declaraciones de componentes y estableciendo vínculos entre ambas clases de servicios.

Darwin soporta la descripción de arquitecturas que se reconfiguran dinámicamente a través de dos construcciones: instanciación tardía [lazy] y construcciones dinámicas explícitas [laxa]. Utilizando instanciación laxa, se describe una configuración y se instancian componentes sólo en la medida en que los servicios que ellos provean sean utilizados por otros componentes. La estructura dinámica explícita, en cambio, se realiza mediante constructores de configuración imperativos. De este modo, la declaración de configuración deviene un programa que se ejecuta en tiempo de ejecución, antes que una declaración estática de la estructura.

Darwin no proporciona una base adecuada para el análisis de la conducta de una arquitectura, debido a que el modelo no dispone de ningún medio para describir las propiedades de un componente o de sus

servicios más que como comentario. Los componentes de implementación se interpretan como cajas negras, mientras que la colección de tipos de servicio es una colección dependiente de plataforma cuya semántica tampoco se encuentra interpretada en el framework de Darwin.

Objetivo principal*:* Como su nombre lo indica, Darwin está orientado más que nada al diseño de arquitecturas dinámicas y cambiantes.

Estilos: El soporte de Darwin para estilos arquitectónicos se limita a la descripción de configuraciones parametrizadas. Un estilo será expresable en Darwin en la medida en que pueda ser constructivamente caracterizado; o sea, para delinear un estilo hay que construir un algoritmo capaz de representar a los miembros de un estilo.

Interfaces: En Darwin las interfaces de los componentes consisten en una colección de servicios que pueden ser provistos o requeridos.

Conectores: En Darwin no es posible ponerle nombre, sub-tipear o reutilizar un conector. Tampoco se pueden describir patrones de interacción independientemente de los componentes que interactúan.

Semántica: Darwin proporciona una semántica para sus procesos estructurales mediante el cálculo. Cada servicio se modela como un nombre de canal, y cada declaración de enlace (binding) se entiende como un proceso que trasmite el nombre de ese canal a un componente que requiere el servicio. Este modelo se ha utilizado para demostrar la corrección lógica de las configuraciones de Darwin. Dado que el cálculo ha sido designado específicamente para procesos móviles, su uso como modelo semántico confiere a las configuraciones de Darwin un carácter potencialmente dinámico.

Análisis y verificación: A pesar del uso de un modelo de cálculo para las descripciones estructurales, Darwin no proporciona una base adecuada para el análisis del comportamiento de una arquitectura. Esto es debido a que el modelo no posee herramientas para describir las propiedades de un componente o de los servicios que presta. Las implementaciones de un componente vendrían a ser de este modo cajas negras no interpretadas, mientras que los tipos de servicio son una colección dependiente de la plataforma cuya semántica también se encuentra sin interpretar en el framework de Darwin.

Interfaz gráfica: Darwin proporciona notación gráfica. Existe también una herramienta gráfica: Asistente

de arquitectura de Software (SAA por sus siglas en inglés) que permite trabajar visualmente con lenguaje Darwin. El desarrollo de SAA parecería estar discontinuado y ser fruto de una iniciativa poco formal, lo que sucede con alguna frecuencia en el terreno de los ADLs debido a su poca madurez.

Soporte de lenguajes: Darwin soporta desarrollos escritos en C++, aunque no presupone que los componentes de un sistema real estén programados en algún lenguaje en particular.

Observaciones: Darwin carece de la capacidad de definir nuevos tipos, soportando sólo una amplia variedad de tipos de servicio predefinidos. Darwin presupone que la colección de tipos de servicio es suministrada por la plataforma para la cual se desarrolla una implementación, y confía en la existencia de nombres de tipos de servicio que se utilizan sin interpretación, sólo verificando su compatibilidad.

Disponibilidad de plataforma: Aunque este ADL fue originalmente planeado para ambientes tan poco vinculados al modelado corporativo como hoy en día lo es Macintosh, en Windows se puede modelar en lenguaje Darwin utilizando SAA. Esta aplicación requiere Ambiente de Ejecución de Java (JRE).

2.3.6 Rapide.

Se puede caracterizar como un lenguaje de descripción de sistemas de propósito general que permite modelar interfaces de componentes y su conducta observable. Sería tanto un ADL como un lenguaje de simulación. La estructura de Rapide es sumamente compleja, y en realidad articula cinco lenguajes: el lenguaje de tipos describe las interfaces de los componentes; el lenguaje de arquitectura describe el flujo de eventos entre componentes; el lenguaje de especificación describe restricciones abstractas para la conducta de los componentes; el lenguaje ejecutable describe módulos ejecutables; y el lenguaje de patrones describe patrones de los eventos. Los diversos sub-lenguajes comparten la misma visibilidad, alcance y reglas de denominación, así como un único modelo de ejecución.

Objetivo principal*:* Simulación y determinación de la conformidad de una arquitectura.

Interfaces: En Rapide los puntos de interfaz de los componentes se llaman constituyentes.

Conectores*:* Siendo lo que (36) llama un lenguaje de configuración in-line, en Rapide no es posible poner nombre, sub-tipear o reutilizar un conector.

Semántica – Mientras muchos lenguajes de tipo ADL no soportan ninguna especificación semántica de sus componentes más allá de la descripción de sus interfaces, Wright y Rapide permiten modelar la conducta de sus componentes. Rapide define tipos de componentes (llamados interfaces) en términos de una colección de eventos de comunicación que pueden ser observados (acciones externas) o iniciados (acciones públicas). Las interfaces de Rapide definen el comportamiento computacional de un componente vinculando la observación de acciones externas con la iniciación de acciones públicas. Cada especificación posee una conducta asociada que se define a través de conjuntos de eventos parcialmente ordenados (posets); Rapide utiliza patrones de eventos para identificar posets, de manera análoga a la del método match de las expresiones regulares de .NET Framework. Para describir comportamientos Rapide también implementa un lenguaje cuyo modelo de interfaz se basa en Standard ML, extendido con eventos y patrones de eventos.

Análisis y verificación automática*:* En Rapide, el monitoreo de eventos y las herramientas nativas de filtrado facilitan el análisis de arquitectura. También es posible implementar verificación de consistencia y análisis mediante simulación. En esencia, en Rapide toda la arquitectura es simulada, generando un conjunto de eventos que se supone es compatible con las especificaciones de interfaz, conducta y restricciones. La simulación es entonces útil para detectar alternativas de ejecución. Rapide también proporciona una caja de herramientas específica para simular la arquitectura junto con la ejecución de la implementación.

En lo anteriormente expuesto, se muestra una gran ventaja entre Jacal y los demás ADLs analizados, pues muestra un gran adelanto ya que permite mediante sus animaciones ver cómo funcionará el sistema de acuerdo a la arquitectura planteada. Pero debido a la poca bibliografía existente sobre este lenguaje, lo que afecta la capacitación del personal para el trabajo con el mismo, ligado a que es un software propietario, para la realización del caso de estudio que se realiza más adelante se utilizará el lenguaje ACME, para aprovechar su característica fundamental que esta dada por ser un lenguaje que además de permitir la descripción arquitectónica, sirve como lenguaje de intercambio con otros ADLs.

2.4 Vistas arquitectónicas.

Vista arquitectónica: representa un aspecto parcial de la arquitectura de software que muestra propiedades físicas del sistema, que agrupa componentes y conectores de acuerdo a la funcionalidad del sistema.

De manera general en cualquier sistema se identifican tres tipos de vistas:

2.4.1 Vista modular.

Módulo:

- En informática es un concepto proveniente de la década de entre 1960 y 1970.
- Basado en la noción de unidad de software que provee servicios a través de una interfaz bien definida.
- La manera de modularización suele determinar características como modificabilidad, portabilidad y reutilización.
- Un módulo es una unidad de código que implementa un conjunto de responsabilidades, puede ser además una clase, una colección de clases, una capa o cualquier descomposición de la unidad de código.

En esta vista se identifican las principales unidades de implementación junto a sus relaciones más relevantes. Estas unidades de implementación no son más que los módulos en que se divide el sistema, los cuales proporcionan determinadas funcionalidades representadas por interfaces. Estos módulos pueden relacionarse de varias formas:

- Is-Part-Of: define las relaciones de agregación entre módulos.
- Depends-on: define relaciones de dependencia entre módulos (por ejemplo un módulo que utiliza otro módulo), esta suele utilizarse en las primeras etapas del proceso de diseño, cuando aún no se ha decidido el tipo de dependencia, porque luego puede reemplazarse por una más precisa.
- Is-A: define relaciones de generalización entre un módulo más específico (el hijo) y uno más general (el padre), asumiendo q el hijo es capaz de ser usado en los contextos en que su padre lo es.

Propiedades:

Expresan información importante asociada a cada módulo, así como agregarles restricciones a estos.

Nombre: El nombre del módulo debe sugerir su rol dentro del sistema, así como reflejar su posición dentro de una posible descomposición jerárquica del mismo.

Responsabilidades: Debe definir con suficiente detalle que es lo que el módulo hace en el contexto del sistema.

Visibilidad de las interfaces: Documentar las interfaces contribuye a establecer con precisión el módulo. Un módulo puede no tener interfaces, o puede tener varias, aunque de estas no todas son accesibles desde fuera del contexto.

Análisis: A partir de estas vistas es posible analizar lo siguiente:

Trazabilidad de requerimientos: aquí se analiza como los requerimientos funcionales son soportados por las responsabilidades de los distintos módulos

Análisis de Impacto: es lo que nos permite predecir el efecto de modificación del sistema.

Comunicación: estas vistas pueden ser utilizadas para explicar las funcionalidades del sistema a alguien no familiarizado con el mismo, así como que posee distintos niveles de granularidad, presentando una descripción top-down de las responsabilidades del sistema.

Dificultades: no debe utilizarse este tipo de vistas para realizar inferencias sobre el comportamiento del sistema durante su ejecución, dada su naturaleza no es de mucha utilidad para la realización de análisis de performance, confiabilidad u otras características asociadas al tiempo de ejecución.

Relaciones con otros tipos de vista:

Las vistas de módulo pueden ser fácilmente mapeadas a vistas de componentes y conectores, ya que los módulos pueden ser mapeados con los componentes que se ejecutan en tiempo de ejecución, este mapeo puede ser simple como una relación uno a uno, o mucho más complejos, porque fragmentos de módulo pueden corresponderse con fragmentos de componentes.

2.4.2 Vistas de componentes y conectores.

En esta vista se describe el comportamiento y estructura del sistema en tiempo de ejecución. Está formada como su nombre lo indica por componentes (unidades de procesamiento y almacenamiento) y conectores (mecanismos de interacción entre componentes). Su uso propicia un mayor rendimiento, disponibilidad, tolerancia a fallos y seguridad.

Utilidad:

Brinda una vista sobre las entidades de ejecución en acción.

Cada tipo de componente y conector puede presentar varias instancias en el mismo modelo.

Los mecanismos de interacción son elementos de primera clase.

Análisis de atributos de calidad:

Las propiedades del sistema en general pueden ser inferidas a partir de analizar este tipo de diagramas, además que conociendo valores cuantitativos de los atributos de los componentes y conectores se pueden calcular atributos del sistema en su conjunto.

Elementos:

Son entidades con manifestación runtime que consumen recursos de ejecución y contribuyen al comportamiento en ejecución del sistema.

La configuración del sistema es un grafo conformado por la asociación entre componentes y conectores.

Relaciones:

La relación que se aplica es Attachment: la cual indica qué componentes están vinculados con qué conectores, de una manera formal siempre se asocian puertos de componentes con puertos de conectores.

Como una guía de las relaciones se tiene que:

Se debe indicar claramente a que estilo se refiere o de lo contrario indicar una especie de leyenda de tipos de componente y conector.

Vincular un conector solo a un puerto específico.

Dejar clara la validez del vínculo, de no ser así justificarlo.

Indicar cuales puertos son usados para conectar el sistema con su entorno externo.

Confiabilidad: Puede usarse para determinar la funcionalidad del sistema en su conjunto.

Performance:

-Tiempo de respuesta / carga.

-Tiempo de latencia y volumen de procesamiento.

Recursos requeridos:

-Necesidad de almacenamiento.

-Necesidad de procesamiento.

Propiedades:

-Funcionalidad: Funciones mapeadas sobre el componente.

-Protocolos: Patrones de eventos o acciones que pueden tener lugar en una iteración representado por el elemento.

Seguridad: Encripta, Audita y Autentica.

Utilidad:

Estas son algunas de las preguntas que se pueden responder con el uso de esta vista:

-¿Cuáles son los componentes ejecutables y cómo interactúan?

-¿Cuáles son los repositorios y a qué componentes acceden?

-¿Qué partes del sistema son replicadas y cuántas veces?

-¿Cómo progresan los datos a lo largo del sistema mientras este se ejecuta?

-¿Qué protocolos de interacción son usados por las entidades comunicantes?

-¿Qué parte del sistema se ejecuta en paralelo?

-¿Cómo la estructura del sistema puede cambiar a medida que se ejecuta?

Para que no se debe ser usado: No se debe usar para modelar elementos de diseño que no tienen comportamiento runtime, además es necesario dejar claro que una clase no es un componente, estos no representan de ninguna manera una visión estática del diseño y estar atento a que si no tiene sentido caracterizar la interfaz de un elemento, probablemente sea porque no es un componente.

Relación con otros Tipos de Vista:

Un componente se relaciona al menos con un módulo y este último a su vez puede estar relacionado con varios componentes.

2.4.3 Vistas de asignación.

Esta vista describe las relaciones entre los elementos de las vistas modulares y otros aspectos del entorno del sistema, está formada por elementos de las vistas modulares y elementos del entorno (hardware, recursos humanos, ficheros, etc.).

En pos de construir el sistema, se debe resolver, como se relaciona la infraestructura de hardware y software con la arquitectura.

Como se relaciona la gestión de configuración con la arquitectura.

Como se relaciona la estructura de asignación de trabajos y el equipo de desarrollo con la arquitectura.

Dicha vista posee fundamentalmente dos tipos de elementos arquitectónicos de la Vista de Módulo o de la Vista de C & C, así como los Objetos del Entorno, estos pueden ser cualquier elemento del entorno del software (Un procesador, un disco, un servidor, un desarrollador, etc.)

La relación de este tipo de vista es la de asignado a (allocated to): La dirección es desde el elemento de software hacia el elemento del entorno, un elemento de arquitectura puede ser asignado a muchos elementos del entorno, y un elemento del entorno puede tener asignados múltiples elementos de software. La asignación puede cambiar con el tiempo, ya sea durante el desarrollo como durante la ejecución, existiendo técnicas para analizar estos escenarios.

Propiedades: La asignación de un elemento de arquitectura a un elemento del entorno implica que las propiedades requeridas son brindadas por el elemento del entorno, en cualquier otro caso la asignación no sería válida.

Utilidad: Sirve como base para realizar distintos análisis que requieran comprender varios aspectos relacionados con el entorno en donde el software se desarrolla o ejecuta, este tipo de vista depende de la existencia de los otros modelos.

Capítulo 3: Características del sistema.

3.1. Introducción.

En este capítulo se hará una propuesta de la arquitectura del proyecto de Gestión de Inventario a través de un Lenguaje de Descripción de Arquitectura (ADL). Para esto se analizarán los principales **escenarios** y artefactos arquitectónicos obtenidos en el sistema antes mencionado resaltando aquellos procesos que merecen ser tratados en un ADL. En este sentido se seleccionó ACME puesto que sus propiedades y características garantizan la totalidad de los objetivos que se desean alcanzar.

Para dar cumplimiento a la idea anterior el presente capítulo se estructura de la siguiente forma:

1. Línea Base de la Arquitectura.
2. Documento de Descripción de la arquitectura.

3.2 Línea Base de la Arquitectura.

La Línea Base de la Arquitectura contiene los principales elementos para lograr la máxima abstracción en el diseño arquitectónico del sistema a desarrollar. Para su desarrollo se decidió aplicar ACME, ADL que se describió en el Capítulo 1, el cual constituye la base que da soporte al presente trabajo puesto que asegura un intercambio entre arquitecturas e integración de ADLs y además por su capacidad intrínseca como ADL puro.

En esta Línea Base de la Arquitectura se expondrán los diferentes estilos arquitectónicos que se utilizarán en el desarrollo del sistema de Gestión de Inventario según José Raúl Perera (37). También se mostrarán los principales componentes o elementos arquitectónicamente significativos como son los conectores y sus configuraciones, así como los distintos patrones arquitectónicos. Igualmente se presentarán las principales restricciones de hardware o software de la arquitectura, así como las tecnologías y herramientas empleadas para el desarrollo del sistema.

3.2.1 Propósito.

El propósito de este documento es representar toda la arquitectura del Sistema Gestión de Inventario, según se obtuvo en su documento de línea base, de una manera más acorde a los objetivos planteados a través de una serie de **vistas estilísticas** y sus respectivas descripciones.

Por otra parte se persigue modelar cada una de las vistas arquitectónicas propuestas por la metodología RUP según se obtuvieron en la línea base original a través del uso de ACME, asegurando la evolución de la misma teniendo en cuenta que esta fue desarrollada sobre UML.

3.2.2 Organigrama de la arquitectura.

La arquitectura de software no es más que la estructura del sistema vista desde distintos niveles de abstracción, que muestra conectores, elementos importantes arquitectónicamente, además de configuraciones y restricciones.

El Sistema de Gestión de Inventario y Almacén, entra en la clasificación de Software de Gestión, este debe almacenar la información en una base de datos, la que contiene la información, que se transforma para facilitar las operaciones comerciales y posibilitar la toma de decisiones en cuanto a las principales operaciones (23).

1. Entrada de productos por compras

2. Movimiento entre secciones (Entrada y salida)

3. Transferencia entre almacenes.

4. Ajustes de Inventarios (por mermas, roturas, etc.)

5. Ubicación de productos

6. Cambio de Código

7. Ventas a clientes terceros

8. Reversión de Operaciones

9. Elaboración de Despieces

10. Elaboración de Escandallos

Además el sistema realizará las operaciones convencionales del procesamiento de datos, lo que hace un poco más densa la lógica de negocio.

Mientras más distribuido esté el código será mucho mejor para el soporte futuro del sistema. Esta característica debe tenerse en cuenta debido a la lógica organizacional de los sistemas de inventarios que pueden cambiar con gran rapidez.

A partir de los elementos mencionados la arquitectura del sistema se ha organizado a partir del estilo de arquitectura “Layers” (Capas).

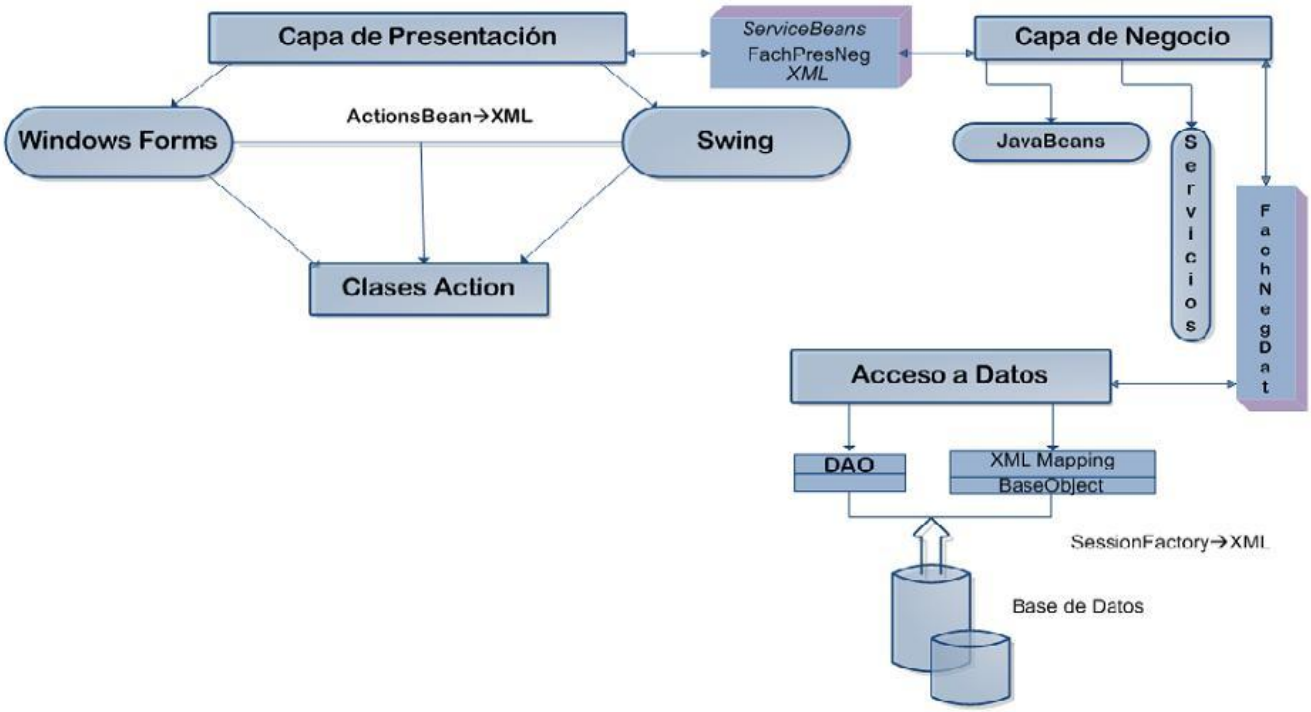

Figura 15. Estructura del estado del Sistema de gestión de Inventarios y Almacenes.

Esta arquitectura muestra como primera capa la capa de presentación, la cual se encuentra dividida en dos subcapas fundamentales que se complementan para permitir al cliente la interacción con la aplicación.

Interfaz de usuario: a través de esta subcapa el usuario logra el intercambio de información con el sistema. Contiene componentes como WindowsForms y algunos componentes desarrollados a partir del framework Swing.

Actions: esta subcapa se encarga de gestionar los eventos y la relación con el modelo de datos que se debe mostrar, además de tratar la validación de los datos de entrada al sistema. Está compuesta principalmente por ActionsBeans.

Además se incluye el patrón Fachada (FachPresNeg), el cual proporciona cierta abstracción para la conexión con la capa de negocio, aportando también su interfaz para la interacción entre estas. Este patrón está compuesto por clases estáticas que solo funcionan como puente de acceso a funcionalidades que es necesario instanciarlas. Esta Fachada se implementa a través del framework Spring aprovechando sus funcionalidades que se expondrán más adelante.

Las 2 subcapas anteriormente mencionadas (Interfaz de usuario y Actions) están conectadas a través de clases Actions, que mediante ActionBeans y XML, crea los JavaBeans que logran integrar las interfaces de usuario con las clases que controlan las acciones.

Como segunda capa de la arquitectura se presenta la capa de **Negocio**, en la cual se encuentra toda la lógica del negocio y se aplican cada una de las reglas de negocio. Para esto se emplean los servicios que son componentes que encapsulan las funcionalidades del negocio y que ejecutan las transacciones, cálculos y devuelven la información que se necesita.

Ejemplo.

```
<bean id="enAutenticarForm"
 class="zun.inventario.administracion.visual.form.autenticar.EnAutenticarForm" parent="baseForm">
    <property name="autenticarAction">
        <ref bean="autenticarAdminAction" />
    </property>
</bean>
```

Este ejemplo muestra como se crea el Bean que corresponde a la WindowsForms de autenticar (enAutenticarForm) y se le referencia la propiedad Bean de la clase Action (autenticarAction) que controla las acciones de esta WindowsForm.

La **capa de acceso a datos** es la encargada de acceder al repositorio de datos y ejecutar la lógica de acceso a datos. Está compuesta por dos subcapas:

1. XML Mapping (ficheros de mapeo)

   Estos ficheros XML son los contenedores de la información Base de Datos a la se hace referencia, que incluyen los campos de cada tabla así como sus relaciones.

2. BaseObject(Objetos de negocio)

   Estos objetos se generan a partir del mapeo de las tablas de la base de Datos y constituyen los objetos con que se trabaja en la aplicación.

3. DAO(Objetos de Acceso Datos)

   Estos objetos acceden a la Base de Datos y realizan las operaciones definidas para poder ejecutar las transacciones.

4. En esta capa también se incluye el patrón Fachada (FachNegDat), para aumentar el nivel de abstracción en la conexión con la capa de negocio, aportando además su interfaz para la interacción entredichas capas. Este patrón está compuesto por clases estáticas que solo funcionan como puente de acceso a funcionalidades que es necesario instanciarlas. Esta Fachada se implementa a través del framework Spring aprovechando sus funcionalidades que se expondrán más adelante.

3.2.3 Conectores / Configuraciones.

No son más que la forma en que se comunican los distintos componentes definidos por el estilo arquitectónico.

Para conectar la capa de Presentación y la capa de Negocio se utiliza ServiceBeans que no son más que las referencias creadas por la capa de Presentación a través de ActionsBeans para utilizar los servicios de la capa de Negocio. Esta referencia puede hacerse a través del constructor o alguna propiedad.

Ejemplo:

```
<bean id="zunBaseAction"
    class="zun.inventario.core.visual.impl.ZunBaseActionImpl">
    <constructor-arg>
        <ref bean="zunService" />
    </constructor-arg>
</bean>
```

Este ejemplo muestra como se referencia al Bean zunBaseAction el Bean del servicio zunService por el constructor, logrando que una vez que se invoque el constructor de la clase Action este invoque a su vez dicho servicio, conectando ambas capas.

En la conexión entre las capas de Negocio y la de Acceso a Datos también se utiliza este servicio pues los DAO que se crean en la capa de Acceso a Datos son referenciados por los Beans creados en la capa de Negocio, accediendo de esta forma a funcionalidades propias de los DAO.

SessionFactory, es un Bean del framework Hibernate que se utiliza para la conexión entre la capa de Acceso a Datos y la Base de Datos, proporcionando la conexión al repositorio de datos.

Ejemplo:

```
<hibernate-configuration>
  <session-factory>
    <property name="connection.driver_class">com.microsoft.jdbc.sqlserver.SQLServerDriver
    <property name="connection.password">mintur</property>
    <property name="connection.url">jdbc:microsoft:sqlserver://10.7.13.23:1433;DatabaseNa
    <property name="connection.username">mintur</property>
    <!-- Settings for a local SQLServer database. -->
    <property name="dialect">org.hibernate.dialect.SQLServerDialect</property>
  </session-factory>
</hibernate-configuration>
```

Como restricción fundamental se tiene la impuesta por este estilo que no es más que el uso de los componentes de las capas inferiores solo por su capa inmediata superior.

3.2.4 Frameworks de desarrollo.

Los frameworks representan otro nivel de abstracción dentro de la descripción de la arquitectura. Estos son espacios de trabajos definidos para un determinado dominio de la aplicación que contiene varios componentes implementados y sus interfaces definidas.

Debido a la estructura en capas que va a presentar el sistema se utilizarán tres frameworks fundamentales, distribuidos en cada capa de la aplicación, los cuales además de tener sus componentes implementados, permiten el desarrollo de nuevos componentes que puedan ser utilizados en el sistema:

- Capa de Presentación: Framework Swing
- Capa de Negocio: Framework Spring 2.0
- Capa de Acceso a Datos: Framework Hibernate 3.2

3.2.4.1 Framework Swing.

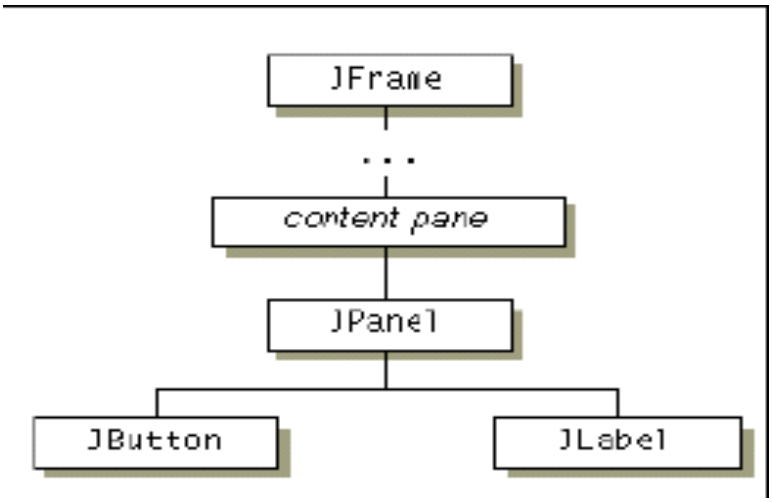

Fig. 16. Ejemplo de Jerarquía de clases de Swing

Características. (38)

- Amplia variedad de componentes: En general las clases que comiencen por "J" son componentes que se pueden añadir a la aplicación. Por ejemplo: JButton.
- Aspecto modificable (look and feel): Se puede personalizar el aspecto de las interfaces o utilizar varios aspectos que existen por defecto (Metal Max, Basic Motif, Window Win32).
- Arquitectura Modelo-Vista-Controlador: Esta arquitectura da lugar a todo un foque de desarrollo muy arraigado en los entornos gráficos de usuario, realizados con técnicas orientadas a objetos. Cada componente tiene asociado una clase de modelo de datos y una interfaz que utiliza. Se puede crear un modelo de datos personalizado para cada componente, con sólo heredar de la clase Model.
- Gestión mejorada de la entrada del usuario: Se pueden gestionar combinaciones de teclas en un objeto KeyStroke y registrarlo como componente. El evento se activará cuando se pulse dicha combinación si está siendo utilizado el componente, la ventana en que se encuentra o algún hijo del componente.
- Objetos de acción (action objects): Estos objetos cuando están activados (Enabled) controlan las acciones de varios objetos componentes de la interfaz.
- Contenedores anidados: Cualquier componente puede estar anidado en otro. Por ejemplo, un gráfico se puede anidar en una lista.
- Escritorios virtuales: Se pueden crear escritorios virtuales o "interfaz de múltiples documentos" mediante las clases JDesktopPane y JInternalFrame.
- Diálogos personalizados: Se pueden crear multitud de formas de mensajes y opciones de diálogo con el usuario, mediante la clase JOptionPane.
- Componentes para tablas y árboles de datos: Mediante las clases JTable y JTree.
- Potentes manipuladores de texto: Además de campos y áreas de texto, se presentan campos de sintaxis oculta JPassword, y texto con múltiples fuentes JTextPane. Además hay paquetes para utilizar ficheros en formato HTML o RTF.
- Soporte a la accesibilidad: Se facilita la generación de interfaces que ayuden a la accesibilidad de discapacitados, por ejemplo en Braille.

Todas las clases componentes de Swing (clases hijas de JComponent), son hijas de la clase Component de AWT.

3.2.4.2 Modelo del framework Swing.

Este framework está compuesto por varios paquetes:

- **Paquete beanInfo**: Contiene las clases que manejan los bean.
- **Paquete event:** Contiene un conjunto de interfaces que posibilitan el control de los eventos de los componentes visuales.
- **Paquete plaf:** Contiene los paquetes que definen las clases con los estilos en que se muestran los componentes visuales.
- **Paquete table:** Contiene el conjunto de clases que permiten crear tablas, estilos de tablas y sus formas.
- **Paquete text:** Contiene las clases necesarias para trabajar los documentos en formato RTF o HTML.
- **Paquete tree:** Contiene las clases e interfaces que permiten el trabajo del componente árbol.

3.2.4.3 Framework Spring 2.0.

Spring es un framework que facilita la creación de diversas aplicaciones. Diseñado en módulos, con funcionalidades específicas y consistentes con otros módulos. Facilita el desarrollo de nuevas funcionalidades y hace que la curva de aprendizaje sea favorable para el desarrollador, al ser fácil de asimilar.

Su objetivo central es permitir la reutilización de los objetos de negocio y acceso a datos, no atados a servicios J2EE específicos.

Filosofía del Framework Spring

- Proveer un framework no invasivo.
- Siempre que se pueda rehusar código.
- Plug and Play de componentes.

Spring provee:

- Una poderosa configuración de la aplicación basada en JavaBeans, aplicando los principios de Inversión de Control (IoC). Esto hace que la configuración de aplicaciones sea rápida y sencilla.
- Ya no es necesario tener singletons ni ficheros de configuración, una aproximación consistente y elegante. Esta factoría de Beans puede ser usada en cualquier entorno, desde el contexto de la aplicación.

Ejemplo:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
                       "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
    <import
        resource="classpath:zun/inventario/commons/model/applicationContext-resources.xml" />

    <import
        resource="classpath:zun/inventario/core/applicationContext-core.xml" />

    <import
        resource="classpath:zun/inventario/nomencladores/applicationContext-nomencladores.xml" />

    <import
        resource="classpath:zun/inventario/administracion/applicationContext-administracion.xml" />

    <import
        resource="classpath:zun/inventario/almacen/applicationContext-almacen.xml" />

    <import
        resource="classpath:zun/inventario/compras/applicationContext-compras.xml" />

    <import
        resource="classpath:zun/inventario/stock/applicationContext-stock.xml" />

</beans>
```

Fichero de configuración del Contexto de la aplicación, importa cada uno de los XML que configuran los módulos de la aplicación.

- Una capa genérica de abstracción para la gestión de transacciones, permitiendo gestores de transacción enchufables (pluggables), y haciendo sencilla la demarcación de transacciones sin tratarlas a bajo nivel. Se incluyen estrategias genéricas y un único JDBC DataSource. En contraste con EJB (Enterprise Java Bean), el soporte de transacciones de Spring no está atado a entornos J2EE.

Ejemplo:

```
<bean
    class="org.springframework.orm.hibernate3.HibernateTransactionManager"
    id="zunTransactionManager">
    <property name="sessionFactory">
        <ref bean="sessionFactoryZUNInv" />
    </property>
</bean>
```

Manejo de transacciones desde el contexto de la aplicación, cada uno de los “Servicios” de la aplicación deben ser instanciados usando un manejador de transacción. Se integra con Hibernate, JDO e iBatis SQL Maps en términos de soporte a implementaciones DAO y estrategias con transacciones. Especial soporte a Hibernate añadiendo convenientes características de IoC, y solucionando muchos de los comunes problemas de integración de Hibernate. Todo ello cumpliendo con las transacciones genéricas de Spring y la jerarquía de excepciones DAO.

- Una capa de abstracción JDBC que ofrece una significativa jerarquía de excepciones (evitando la necesidad de obtener de SQLException los códigos que cada gestor de base de datos asigna a los errores), simplifica el manejo de errores, y reduce considerablemente la cantidad de código necesario.

Ejemplo:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
                       "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>

    <bean id="throwsAdvice"
        class="zun.inventario.core.aop.advice.EventoAfterThrowAdvice" />

    <!-- ZUN ACTION -->
    <bean id="zunAction"
        class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="interceptorNames">
            <list>
                <idref local="throwsAdvice" />
            </list>
        </property>
        <property name="target">
            <ref bean="zunBaseAction" />
        </property>
    </bean>
```

El manejo de las excepciones se trata desde los “aspectos” esto se debe de tejer en cada una de las llamadas al bean zunAction.

Funcionalidad AOP (Programación Orientada a Aspectos), está totalmente integrada en el framework Spring. Se puede aplicar AOP a cualquier objeto gestionado por Spring, añadiendo aspectos como gestión de transacciones declarativa.

- Un framework MVC (Model-View-Controller), construido sobre el núcleo de Spring. Este framework es altamente configurable vía interfaces. De cualquier manera una capa modelo realizada con Spring puede ser fácilmente utilizada con una capa de presentación basada en MVC (Modelo Vista Controlador) como es el caso de Swing. Este modelo se explicó más detalladamente en el Capítulo 1.

3.2.4.4 Framework Hibernate 3.2. (39)

Hibernate es un framework que constituye un motor de persistencia que implementa múltiples funcionalidades. El centro de la arquitectura de Hibernate lo constituyen una serie de interfaces que realizan el grueso de las funcionalidades del framework, dentro de ellas están:

Interfaces	Descripción
Sesión (Session)	Es la más usada en las aplicaciones, es la manejadora de la persistencia, a través de ella se podrán guardar y cargar objetos de la base de datos.
Fabrica de sesiones (Session Factory)	Mediante ella se obtiene la Session.
Configuración (Configuration)	Es usada para configurar Hibernate se utiliza para especificar la ruta de los documentos de mapeo.
Transacción (Transaction)	Se utiliza para el control de las transacciones.
Consultas (Query y Criteria)	Permite la ejecución de consultas a la Base de Datos.

Tabla5. Interfaces del framework Hibernate.

Además Hibernate brinda otras funcionalidades importantes y que serán utilizadas en nuestras aplicaciones, por ejemplo:

- La interface Type, es un elemento fundamental y muy poderoso, permite el mapeo de tipos “java” a varias columnas de la base de datos. Incluye tipos como Calendar, byte [], y permite además definir tipos de datos propios.

- Las interfaces de Llamada de Retorno (Callback) gestionan el ciclo de vida de los objetos (Lifecycle, Validatable).
- Un dialecto orientado a objetos (HQL) fácil de manejar y familiar a SQL que aunque no es un lenguaje de manipulación de datos como este, es usado solamente para extraer datos no para borrar, insertar o actualizar, además permite realizar complejas consultas.

Hibernate puede ser configurado y ejecutado por cualquier aplicación java y entorno desarrollo.

Define dos tipos de configuración:

- Entorno manejado: estos entornos que proveen reservas de recursos como conexiones a base de datos (connections pool), transacciones y seguridad declarativa, en este caso se encuentran los servidores de aplicación como JBoss, BEA WebLogic entre otros.
- Entorno no manejado, es el caso de los contenedores de servlet como Tomcat, las aplicaciones de escritorio también son incluidas aquí; este tipo de entorno no provee transacciones automáticas ni manejos de recursos, la propia aplicación realiza el manejo de las conexiones a base de datos.

Otros aspectos que son esenciales en el desarrollo de aplicaciones con Hibernate lo constituyen los ficheros de mapeo y configuración. Para cada clase persistente se le hace corresponder un fichero de mapeo que es el lugar donde se le especifica al framework como va a persistir dicha clase en la base de datos. El archivo de configuración es el que dice a Hibernate lo referente a la conexión que se puede alcanzar vía JNDI en el caso de los entornos manejados o cargando la conexión con el driver correspondiente al gestor haciendo uso si se quiere de alguna herramienta que maneje la reserva de conexiones todo esto en el mismo archivo.

Ejemplo:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
<hibernate-mapping schema="dbo"
    package="zun.inventario.commons.model.data">
    <class name="AUsuario" table="A_Usuario" schema="dbo"
        optimistic-lock="none">
        <id name="usuario" type="string" unsaved-value="-1">
            <column name="usuario" not-null="true" unique="true"
                index="PK223" />
            <generator class="assigned" />
        </id>
        <many-to-one name="rol"
            entity-name="zun.inventario.commons.model.data.ARol"
            foreign-key="RefA_ROL551" lazy="false">
            <column name="id_Rol" not-null="true" index="PK222" />
        </many-to-one>
        <property name="contrasena" type="string" column="contrasena"
            length="50" />
        <property name="nombreU" type="string" column="nombre_U"
            length="50" />
        <property name="primerApellido" type="string"
            column="primer_Apellido" length="50" />
        <property name="segundoApellido" type="string"
            column="segundo_Apellido" length="50" />
```

Fichero de mapeo correspondiente a la tabla de la Base de Datos A_Usuario.

Ventajas del uso de Hibernate:

- Es un software open source y gratis
- Abstrae el manejo del código SQL, permitiendo ganar tiempo de desarrollo.
- Las clases que persisten no tienen que implementar ninguna interfaz especial ni extender ninguna clase, posibilitando su reutilización en distintas partes de la aplicación con el objetivo de transportar datos.
- Permite el manejo de almacén de conexiones (Connections Pool).
- No necesita ningún entorno especial para correr, dígase servidor, aplicaciones, contenedores, etc.
- Permite manejar las transacciones de manera eficaz.

Desventajas del uso de Hibernate:

- No es un estándar J2EE.
- Es un software complejo por lo que la capacitación del personal para su uso debe garantizarse antes del comienzo del proyecto para poder optimizar tiempo y aprovechar las comodidades del framework.

3.2.5. Lenguaje, tecnología y herramientas de soporte al desarrollo.

3.2.5.1 Java.

Java muestra un gran avance en el entorno de desarrollo de software, esto puede afirmar pues presenta elementos significativos que lo diferencian desde el punto de vista tecnológico de los demás lenguajes:

- Es un lenguaje de programación orientado a objetos con una sintaxis fácilmente accesible, además de contar con un entorno robusto y agradable.
- Ofrece un conjunto de clases potentes y flexibles.

Características de Java

- Orientación a Objetos.
- En Java el concepto de objeto resulta sencillo y fácil de ampliar. Además se conservan elementos "no objetos", como números, caracteres y otros tipos de datos simples.
- Riqueza semántica.

  Pese a su simpleza se ha conseguido un considerable potencial, y aunque cada tarea se puede realizar de un número reducido de formas, se ha conseguido un gran potencial de expresión e innovación desde el punto de vista del programador (40).

- Robusto.

  Java verifica su código al mismo tiempo que lo escribe, y una vez más antes de ejecutarse, de manera que se consigue un alto margen de codificación sin errores, es decir que estos van siendo

detectados en su gran mayoría en tiempo de complicación, ya que Java es estricto en cuanto a tipos y declaraciones, y así lo que es rigidez y falta de flexibilidad se convierte en eficacia. Respecto a la gestión de memoria, sus características liberan a los programadores de una familia entera de errores (la aritmética de punteros), ya que se ha prescindido por completo de estos. Este lenguaje posee una gestión avanzada de memoria llamada recolector de basura (Garbage Collector), y un manejo de excepciones orientado a objetos integrados. Estos elementos realizarán muchas tareas, que antes eran tediosas, a la vez que obligadas para el programador como la destrucción de objetos (40).

- Fácil aprendizaje.

  Java es más complejo que un lenguaje simple, pero más sencillo que cualquier otro entorno de programación. El único obstáculo que se puede presentar es conseguir comprender la programación orientada a objetos, aspecto que, al ser independiente del lenguaje, se presenta como insalvable (40).

- Completado con utilidades.

  El paquete de utilidades de Java contiene un conjunto completo de estructuras de datos complejas y sus métodos asociados, que serán de gran ayuda para implementar applets y otras aplicaciones más complejas. Dispone también de estructuras de datos usuales, como pilas y tablas hash, como clases ya implementadas.

  El JDK (Java Development Kit) suministrado por Sun Microsystems incluye un compilador, un intérprete de aplicaciones, un depurador en línea de comandos, y un visualizador de applets, entre otros elementos.

- Arquitectura neutral.

  Java está diseñado para que un programa escrito en este lenguaje sea ejecutado correctamente independientemente de la plataforma en la que se esté actuando (Macintosh, PC, UNIX…). Para conseguir esto utiliza una compilación en una representación intermedia que recibe el nombre de códigos de byte, que pueden interpretarse en cualquier sistema operativo con un intérprete de Java.

La desventaja de un sistema de este tipo es el rendimiento; sin embargo, el hecho de que Java fuese diseñado para funcionar razonablemente bien en microprocesadores de escasa potencia, unido a la sencillez de traducción a código máquina hacen que Java supere esa desventaja sin problemas (40).

- Seguridad.
  Debido a la gran navegación de códigos malignos (virus, troyanos y demás programas hechos con malas intensiones)a través de Internet, Java presta una atención especial al tema de la seguridad, consiguiendo cierta inmunidad pues un programa desarrollado en Java no puede realizar llamadas a funciones globales ni acceder a recursos arbitrarios del sistema .

Niveles de seguridad de Java:

- Fuertes restricciones al acceso a memoria, como son la eliminación de punteros aritméticos y de operadores ilegales de transmisión.
- Rutina de verificación de los códigos de byte que asegura que no se viole ninguna construcción del lenguaje.
- Verificación del nombre de clase y de restricciones de acceso durante la carga.
- Sistema de seguridad de la interfaz que refuerza las medidas de seguridad en muchos niveles.

En Java se pueden desarrollar programas de tipo:

- Aplicaciones: Se ejecutan sin necesidad de un navegador.
- Applets: Se pueden descargar de Internet y se observan en un navegador.
- JavaBeans: Componentes software Java, que se puedan incorporar gráficamente a otros componentes.

3.2.5.2 IDE (Entorno de Desarrollo Integrado)

Eclipse: Es una plataforma de software de Código abierto independiente de una plataforma para desarrollar, lo que el proyecto llama "Aplicaciones de Cliente Enriquecido", opuesto a las aplicaciones "Cliente-liviano" basadas en navegadores. Esta plataforma, ha sido usada para desarrollar un IDE, como

el llamado Java Development Toolkit (JDT) y el compilador (ECJ) que se embarca como parte de Eclipse (y que son usados también para desarrollar el mismo Eclipse). Sin embargo, también se puede usar para otros tipos de aplicaciones, como es el cliente BitTorrent Azureus.

3.2.5.3 Arquitectura.

La base para Eclipse es la Plataforma de Cliente Enriquecido RCP (Rich Client Platform) dotada de los siguientes componentes:

- Plataforma principal - inicio de Eclipse, ejecución de plugins.
- OSGi - una plataforma para bundling estándar.
- El Standard Widget Toolkit (SWT) - UN widget toolkit portable.
- JFace - manejo de archivos, manejo de texto, editores de texto.
- El Workbench de Eclipse - vistas, editores, perspectivas, asistentes.

El IDE de Eclipse utiliza módulos (plug-in) para proveer todas sus funcionalidades al frente de la Plataforma de Cliente Enriquecido (RCP), a diferencia de otros entornos monolíticos donde las funcionalidades están todas incluidas, las necesite el usuario o no. Este mecanismo de módulos es una plataforma ligera para componentes de software. Eclipse además puede extenderse usando otros lenguajes de programación como son C/C++ y Phytón, que permite a Eclipse trabajar con lenguajes para marcado de texto como LaTeX, aplicaciones en red como Telnet y Sistemas de Gestión de Base de Datos(SGBD).

La definición que da el proyecto Eclipse acerca de su software es: "una especie de herramienta universal - un IDE abierto y extensible para todo y nada en particular".

El SDK de Eclipse incluye las herramientas de desarrollo de Java, ofreciendo un IDE con un compilador de Java interno y un modelo completo de los archivos fuente de Java. Esto permite técnicas avanzadas de refactorización y análisis de código.

Características

En el desarrollo del Sistema de Gestión de Inventario se utilizará Eclipse, versión 3.2.2 que proporciona comodidades como:

- Editor de texto
- Resaltado de sintaxis
- Compilación en tiempo real
- Pruebas unitarias con JUnit
- Control de versiones con CVS
- Integración con Ant
- Asistentes (wizards): para creación de proyectos, clases, tests, etc.
- Refactorización

Asimismo, a través de "plugins" libremente disponibles es posible añadir:

- Control de versiones con Subversión, vía Subclipse.
- Integración con Hibernate, vía Hibernate Tools.
- Integración con Spring, vía Spring Framework.

3.2.5.4 Eclipse como Herramienta de Desarrollo.

- El editor de Eclipse resalta el código Java de manera que sea más fácil leerlo, editarlo y auto completarlo mientras se escribe, como cualquier otro editor Java.
- Una de las ventajas de Eclipse es que compila el código de forma incremental, mientras se escribe, lo que permite detectar errores de compilación en tiempo de programación. También ofrece la posibilidad de subsanar el error automáticamente, eligiendo entre varias posibles soluciones. Además, detecta errores de sintaxis cuando por ejemplo falta un paréntesis o un corchete.

3.2.5.5 Herramientas de refactorización y de código.

- Eclipse incluye herramientas de refactorización, las que permite renombrar métodos, atributos, variables o clases, cambiando todas las referencias que haya en el proyecto hacia el elemento. También permite cambiar la localización de una clase o paquete actualizando las referencias.

- Genera plantillas para la documentación en formato Javadoc para clases, métodos o atributos, con plantillas configurables.
- Mediante la externalización de cadenas, extrae las cadenas de una clase a un fichero de propiedades, que luego puede usarse para traducir la aplicación.

3.2.5.6 Integración con Junit.

- El JDT incluye un plug-in para ejecutar pruebas JUnit en el entorno de programación. Una vista, similar al visor Swing de JUnit que muestra los resultados de los tests.

3.2.5.7 Plug-ins para Eclipse.

Eclipse se distingue en gran medida por mostrar una gran modularidad lo que permite lo que permite añadir plug-ins, personalizando el entorno de trabajo de acuerdo a las necesidades del usuario.

Para el desarrollo del Sistema de Gestión de Inventario se utilizarán algunos plug-ins que no están en la distribución de Eclipse como:

- WindowsBuilder: plug-ins que se utiliza para construcción de interfaces gráficas.
- Hibernate Tool: plug-ins que facilita el trabajo de los desarrolladores pues partiendo de un esquema de base de datos, mediante un proceso de ingeniería inversa permite crear ficheros de configuración para Hibernate yEJB3, componentes Sean y DAO's, además de la realización de consultas interactivamente a las bases de datos a través de HQL y EJB QL3.
- Junit: framework que hace posible la ejecución de clases Java de manera controlada es decir con él se puede evaluar el funcionamiento de los métodos de dichas clases y verificando el comportamiento esperado. También puede controlar las pruebas de regresión que se realizan cuando un parte del código se ha modificado y se desea comprobar que cumple con los requerimientos anteriores sin alterar los resultados. Incluye formas de ver los resultados(runners) que pueden ser de modo texto o gráfico(AWT o Swing)
- SVNeclipse: es un software que se utiliza en el sistema de control de cambios. Una característica importante de SVNeclipse es que, a diferencia de CVS, los archivos versionados no tienen cada uno su 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.

Características

1. Un repositorio remoto donde se alojan las copias y revisiones comunes.
2. A la vez permite mantener una copia local donde se trabaja de forma independiente, cuando se cree conveniente, la copia se envía al repositorio remoto y el sistema.
3. Verifica que no existen conflictos entre las diferentes copias.
4. Puede ser servido, mediante Apache, sobre WebDAV/DeltaV.
5. Maneja eficientemente archivos binarios (a diferencia de CVS que los trata internamente como si fueran de texto).
6. AcmeStudio: este plug-in permite el uso del ACME como Lenguaje de Descripción de Arquitectura para el diseño del sistema desde el punto de vista arquitectónico.

3.2.5.8 Sistema Gestor de Base de Datos (SGBD).

El Sistema de Gestión de Inventario como software de gestión que es, necesita repositorio de información donde almacenar los datos para realizar las distintas operaciones. Como en la aplicación se utilizó el estilo de Arquitectura en Capas, el cual permite una mayor abstracción de la lógica de negocio, en el almacenamiento de datos, la capa de acceso a datos contiene toda la capacidad de acceso como son las consultas y procedimientos almacenados, dejando a la Base de Datos como simple almacén de datos sin ninguna lógica. Para esta aplicación se puede utilizar cualquier Sistema Gestor de Base de Datos, teniendo como única restricción que pueda implementar bases de datos relacionales.

El cambio en el SGBD no costaría un esfuerzo en el desarrollo del sistema, sino que sería el cambio de las clases mapeadas (XML Mapping) en la Capa de Acceso a Datos. Además se implementaría los mismos disparadores (Triggers) para mantener la seguridad y la confiabilidad en los datos, y los roles de usuario para cada una de las opciones.

3.3 Documento Descripción de la Arquitectura.

3.3.1 Introducción.

El Documento de Descripción de la Arquitectura es un documento muy importante en la documentación del desarrollo de un sistema. En él que plasmado de forma general como va a funcionar el sistema, las características principales del sistema así como los requerimientos funcionales y no funcionales que debe cumplir.

3.3.2 Propósito.

El objetivo del Documento Descripción de la Arquitectura es mostrar una visión global del sistema, mediante distintas vistas para mostrar diferentes aspectos del sistema. En el quedan plasmadas las decisiones arquitectónicamente significativas que se tomarán para el desarrollo del sistema. Además brinda una mejor comprensión del sistema, incrementa la organización del desarrollo, fomenta la reutilización y ayuda en la evolución del desarrollo del sistema.

3.3.3 Alcance.

Propicia la toma de decisiones en la arquitectura del Sistema de Gestión de Inventario.

3.3.4 Metas y restricciones arquitectónicas.

3.3.4.1 Requerimientos de hardware.

Estaciones de Trabajo

1. Tener periféricos Mouse y Teclado.

2. Tarjeta de red.

3. 64 Mb de memoria RAM.

Servidor Local

1. Tener periféricos Mouse y Teclado.

2. 1Mb de cache L2, 256 Mb de memoria RAM.

3. 2 GB de espacio libre en disco.

4. Tarjeta de red.

Servidor de herramientas y software

Este estará compuesto por aquellas herramientas que se utilizan en las estaciones de trabajo y que en caso que sufran algún daño por causa de virus o ataque informático el encargado del mantenimiento de las estaciones de trabajo pueda acceder a este servidor y mantener siempre en funcionamiento optima cada estación.

Servidor de correo:

Por esta vía se pueden enviar los reportes de las acciones realizadas a la empresa central sin la necesidad trasladarse hasta esta en persona.

3.3.4.2 Requerimientos de Software.

Estaciones de Trabajo

1. Sistema Operativo: Windows 98 o superior, Linux, Unix.

2. Java Virtual Machine (Máquina Virtual de Java) versión 1.6.

Servidor Local

1. Sistema Operativo: Windows 98 o superior, Linux, Unix.

2. Gestor de Base de Datos: SQL Server 2000.

3. Maquina Virtual de Java versión 1.6.

3.3.4.3 Redes.

1. La red debe soportar la transacción de paquetes de información de al menos 10 máquinas a la vez.

2. La aplicación debe de estar protegida contra fallos de corriente y de conectividad, para lo que se deberá parametrizar los tiempos para realizar copias de seguridad. Implementar las transacciones de paquetes en la red con el protocolo TCP/ IP que permite la recuperación de los datos.

3. La red estará representada por la topología de estrella, en la cual los periféricos instalados en el servidor serán compartidos para todas las estaciones conectadas a este. Se escoge este tipo por su flexibilidad a la hora de aumentar el número de equipos conectados en ella, además que si falla un equipo de la red no se afectará ningún otro a no ser que se dañe el servidor central. El servidor de la red puede ser una computadora basada en el procesador 80386. Esta topología muestra como requisitos una tarjeta de interfaces, cable de 2 hilos sin blindaje y un distribuidor central (HUB).

3.3.4.4 Seguridad.

1. Se utilizarán las reglas de “programación segura”, además de un fuerte tratamiento de excepciones.

2. La tecnología Java utilizada para el desarrollo de la aplicación aporta gran seguridad para el sistema.

3. Para garantizar la seguridad de la Base de Datos, no se permitirá que nadie tenga privilegios de administración sobre la base de datos.

4. Uso de disparadores (Triggers) en la Base de Datos para no permitir la manipulación de los datos directamente en el SGBD.

5. Se deberá utilizar usuarios de base de datos con roles bien definidos para cada una de las operaciones del sistema.

6. La asignación de usuarios y sus opciones sobre el sistema se garantizará desde el modulo de Administración del sistema.

7. Debe existir constancia de quién, desde donde, y cuando se realizó una operación determinada en el sistema.

3.3.4.5. Portabilidad, escalabilidad, reusabilidad.

1. El sistema deberá poder ser utilizado desde cualquier plataforma de software

(Sistema Operativo).

2. El sistema deberá hacer un uso racional de los recursos de hardware de la máquina.

3. El sistema deberá implementar la forma de adaptarse ante cambios en las condiciones económicas del país, que propicien la toma decisiones de nuevas decisiones en la empresa.

4. Deberá implementar servicios que atiendan el pedido de información de otros sistemas que en el futuro la necesiten.

5. Se desarrollará cada pieza del sistema en forma de componentes (elementos) con el fin de reutilizarlos para futuras versiones del sistema.

6. La documentación de la arquitectura deberá ser reutilizable para poder documentarla como una familia de productos.

3.3.4.6 Restricciones de acuerdo a la estrategia de diseño.

1. El diseño de las aplicaciones se hará utilizando la Programación Orientada a Objetos (POO). Encapsulación de la lógica por clases.

2. Se utilizará las tecnologías que brindan los frameworks definidos para cada una de las capas de la aplicación:

- Para la capa de presentación: Swing, aplicación de escritorio.
- Para la capa de lógica del negocio: los objetos del negocio, Framework Spring, la IoC y la programación Orientada a Aspectos.
- Para la capa de Acceso a Datos: Framework Hibérnate.

3.3.4.7. Herramientas de desarrollo.

1. Para modelado se utilizará el plug-in ACMEStudio como Lenguaje de Descripción de Arquitectura (ADL).
2. Para implementación como IDE de desarrollo Eclipse 3.2 el cual demanda como mínimo 768 MB de RAM y recomendado 1.0 GB de RAM al integrarlo con los plugins necesarios.
3. Para implementación los plugins serán:
   - WindowsBuilder, plugin que garantiza el trabajo con componentes visuales y reutilizables para la capa de presentación. Implementa el Framework Swing.
   - Opciones de Spring, para implementar el Framework Spring 2.0.
   - Hibernate Tool el cual facilita la ingeniería inversa y reversa de los ficheros de mapeo entre clases java y tablas de Base de Datos.
   - JUnit el cual facilita las pruebas de unidad.
   - SVNeclipse el cual facilita la integración con Subversión.
4. Como servidor de Base Datos SQL Server 2000. (256 MB de RAM).
5. Como servidor de control de versiones Subversión.

3.3.4.8 Estructura del equipo de desarrollo.

El grupo de desarrollo está dividido en equipos y módulos. Los equipos representan la dirección del proyecto pues garantizan la planificación, formación, gestión de configuración y calidad del proyecto. Además del equipo encargado de la base de datos y el de arquitectura. Los módulos representan las principales funcionalidades del sistema y que serán desarrollados en conjunto por los analistas, programadores y revisores técnicos de cada módulo para lograr mayor independencia en el trabajo.

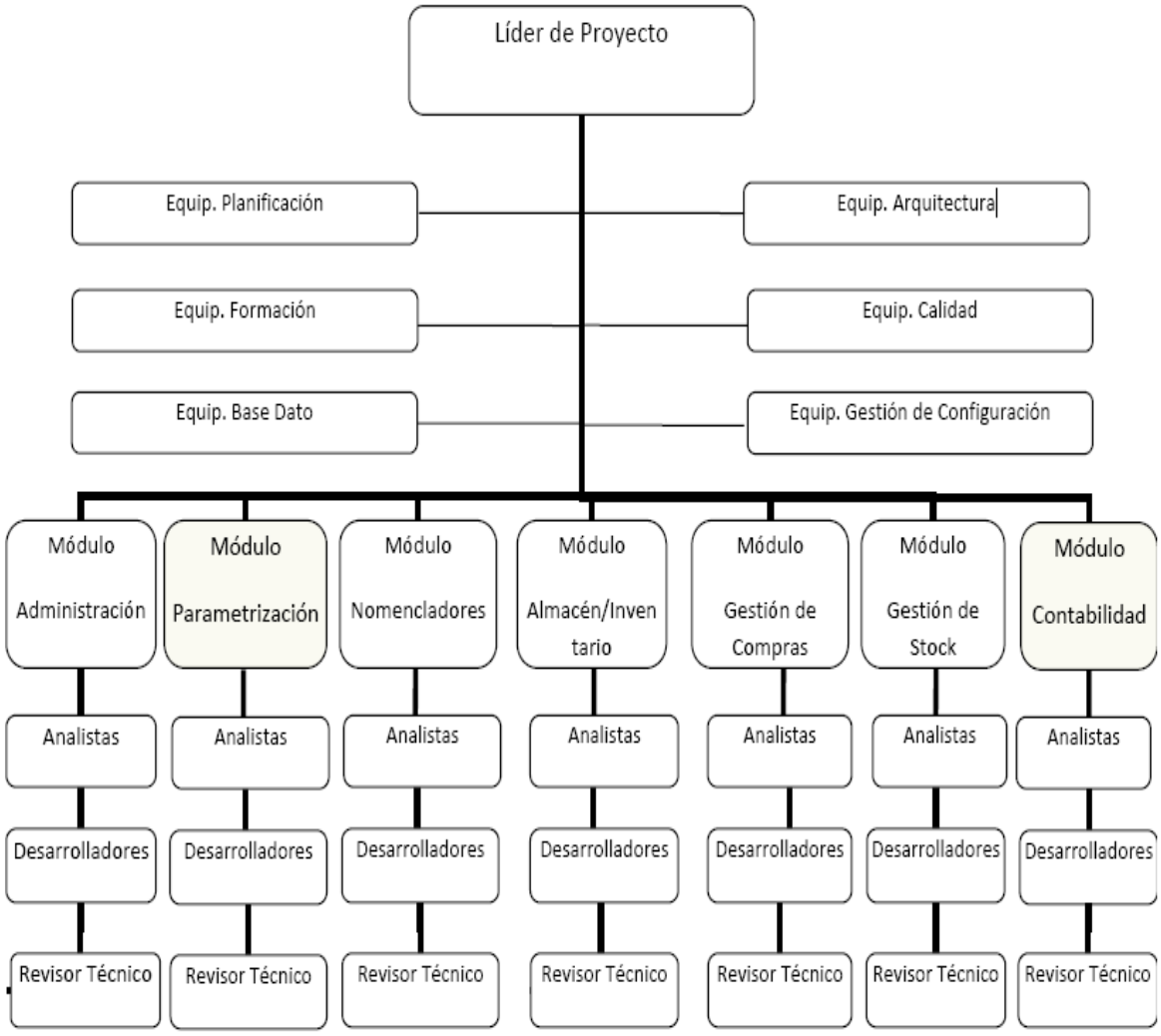

Figura 17. Estructura del equipo de Desarrollo.

Distribución de los puestos de trabajo por roles:

- Rol de analista:
  1. PC con mouse y teclado.
  2. Paquete Office para la documentación del sistema.
  3. Instalación Suite de Racional 2003, para modelar el sistema.
- Rol desarrollador:
  1. Eclipse con los plug-ins necesarios.

2. PC con Mouse, teclado, 1GB de memoria RAM.
3. Instalación de la Máquina Virtual de JAVA (JDK 1.6).

- Rol de revisor técnico:
  1. Paquete Office para la documentación del sistema.
  2. PC con Mouse y teclado.
  3. Instalación Suite de Racional 2003 para modelar los casos de prueba.

3.4 Vistas Arquitectónicas del Sistema de Gestión de Inventario.

3.4.1 Vista de Módulos

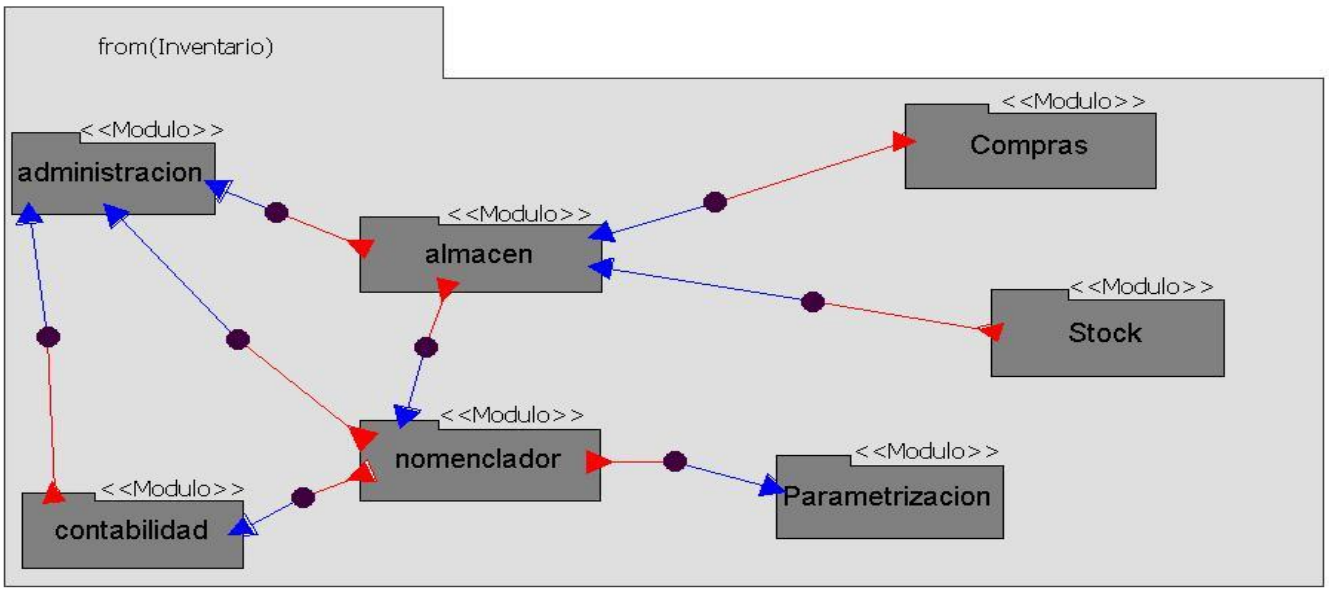

Figura 28. Representación de los Módulos del Sistema de Gestion de Inventario

Estos módulos van a estar compuestos por diferentes casos de uso como son:

1. Módulo de administración: contiene a los casos de uso Gestión de usuarios, Gestión de roles y Gestión de terminales.
2. Módulo Parametrización: contiene los casos de uso de configuración de estructura de la organización, la configuración de los comprobantes contables, dígitos de los códigos entre otros.

3. Módulo Nomencladores: contiene los casos de uso nomenclar producto, nomenclar sección y nomenclar la unidad de medida.
4. Módulo Almacén/Inventario: contiene los casos de uso donde se realizan las principales operaciones del sistema, como son: entrar productos de compra, solicitar producto, rebajar productos por movimientos entre otros.
5. Módulo Stock: contiene los casos de uso que permiten la gestión de Stock de la organización.
6. Módulo Compras: contiene los casos de uso que permiten la gestión de compras y sus procesos derivados.
7. Módulo de Contabilidad: Contiene los casos de uso que permiten la configuración de los parámetros fundamentales para realizar las operaciones como la validación contable y la generación de comprobantes.

3.4.2 Vista de componentes y conectores

Al modelar o describir la arquitectura de un sistema con ADLs, la vista de Casos de Uso de RUP pasa a formar parte de la Vista Modular, debido a que los ADLs solo poseen: componentes, conectores, configuraciones y restricciones, los Actores y Clases de la Vista de Caso de Uso desaparecen quedando modelado con Acme de la siguiente forma:

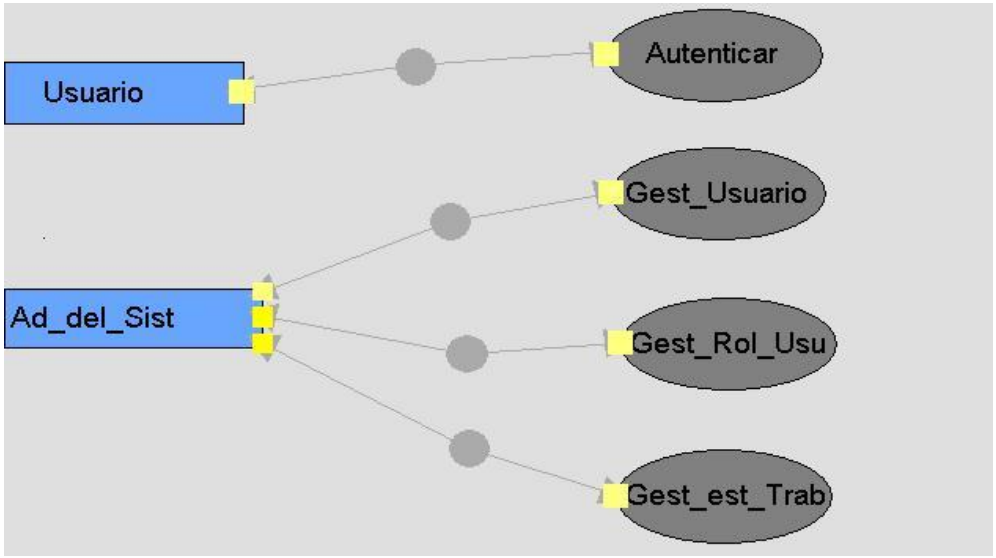

Figura 39. Representación del Caso de Uso Gestionar Usuario

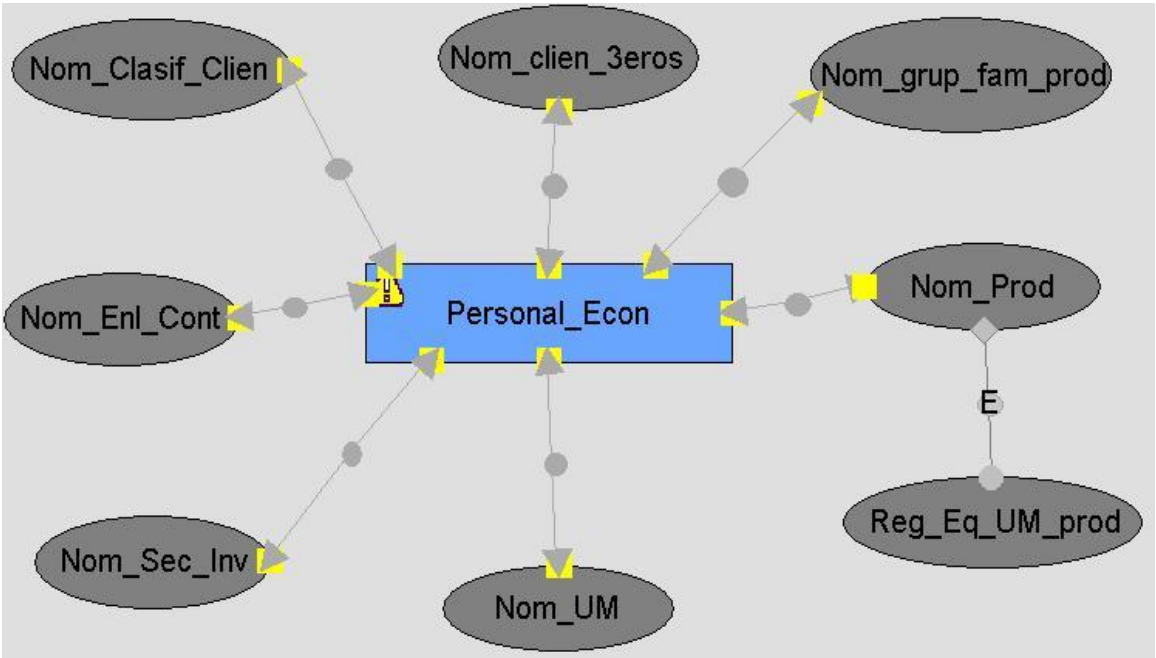

Figura 20. Representación de la relación entre los componentes del sistema y el usuario.

La figura muestra el intercambio de información entre el usuario y los diferentes componentes del Sistema de Gestión de Inventario, intercambio necesario para la satisfacción del cliente.

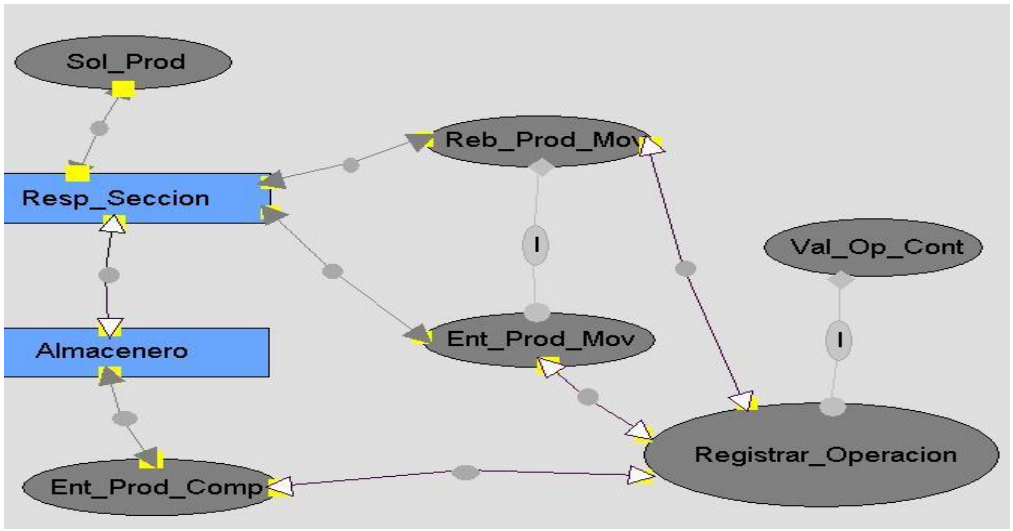

Figura 4 Representación de la relación entre los distintos Casos de Uso del Módulo de inventario y el almacenero (usuario).

3.5 Conclusiones.

La Arquitectura propuesta tiene su base en el desarrollo de componentes y conectores, se ha descrito utilizando ACME para modelar los diferentes Artefactos, así como los Casos de Usos críticos que eran necesarios representar. Además de desarrollarse en él, los Documento de Línea Base y de Descripción de la Arquitectura.

Conclusiones

El presente estudio ha procurado describir herramientas disponibles para el desempeño de la descripción arquitectónica de una forma más eficiente. Los ADLs son convenientes, pero no han demostrado aún ser imprescindibles. La demanda que los ha generado se ha instalado como un tópico permanente de la arquitectura de software, y por eso ha sido útil elaborar este examen.

- Se obtuvo como resultado una serie de aportes en el campo de la Arquitectura de Software, principalmente de carácter investigativo.
- Se propuso un ADL que puede ser utilizado de manera provechosa tanto en la Facultad como en la Universidad logrando una mejora en el Proceso de Desarrollo de Software.
- Se demostró la superioridad de los ADL en el trabajo de descripción de Arquitecturas de Software con un alto nivel de complejidad desde el punto de vista estructural.
- Se propone Acme que dadas sus características y los estudios sobre él, resultó ser el más apropiado para cumplir los propósitos trazados, abriendo simultáneamente multitud de posibles líneas de investigación y desarrollo.
- Se describió de manera exitosa la Arquitectura del Sistema de Gestión de Inventario y Almacén utilizando Acme, incorporando a su anterior descripción nuevos elementos para favorecerla.

Recomendaciones

Con el propósito de mejorar lo logrado en este trabajo se recomienda:

- Continuar con el estudio de los ADL profundizando en Acme y Jacal, dadas sus características y así lograr una mejora en la productividad y calidad del proceso de Desarrollo de Software.
- Aplicar al Sistema de Gestión de Inventario y Almacén los métodos de evaluación de la arquitectura para identificar las posibles debilidades.
- Extender el uso de herramientas de descripción de arquitectura en los proyectos de la facultad para lograr un mayor entendimiento entre los arquitectos de dichos proyectos.

Bibliografía Referenciada

1. **Trujillo Casañola, Yaimí.** *Evaluación teórica de la adopción del enfoque de Factorías de Software en la Universidad de las Ciencias Informáticas.* Universidad de las Ciencias Informáticas. : s.n., 2006.

2. **OSF () OSF DCE Application Development Guide., Cambridge, MA (Estados Unidos).** Open Software Foundation. *Open Software Foundation.* [En línea] 1994. [Citado el: 13 de 2 de 2008.] http://www.Open Software Foundation.com.

3. **OMG.** *The Common Object Request Broker: Architecture and Specification.* 1999.

4. **Vinoski, S.** *New Features for CORBA 3.0. Communications of the ACM.* 1998.

5. **Rogerson, D.** *Inside COM. M. Press.* 1997.

6. **Wesley, Addison y Box, D.** *Essential COM .* 1998.

7. **Microsystems, S.** *JavaBeans API Specification 1.01.* 1997.

8. **Microsoft.** *DCOM - The Distributed Component Object Model.* 1996.

9. **Fayad, M, Schmidt, D y Johnson, R E.** *Application Frameworks, Wiley & Sons. 3.* 1999.

10. **Shaw, M y Garlan, D.** *Software Architecture. Perspectives of an Emerging Discipline.* s.l. : Prentice Hall, 1996.

11. **Steve, Vestal.** *"A cursory overview and comparison of four Architecture Description".* s.l. : Technical Report, Honeywell Technology Center, 1993.

12. **Wolf, Alexander.** *"Succeeding of the Second International Software Architecture Workshop".* s.l. : ACM SIGSOFT Software Engineering Notes, 1997.

13. **Luckham, David y Vera, James.** *An Event-Based Architecture Definition Language.* s.l. : IEEE Transactions on Software Engineering, 1995.

14. **Shaw, Mary, y otros.** *Abstractions for Software Architecture and Tools to Support Them.* s.l. : IEEE Transactions on Software Engineering, 1995.

15. *A classification and comparison framework for software Architecture Description Languages.* **Medvidovic, Neno.** 1996.

16. **Gamma, E.** *Patrones de Diseño.* s.l. : Addison-Wesley, 2002.

17. **Palos, Juan Antonio.** *Catálogo de Patrones de Diseño J2EE . Y II: Capas de Negocio y de Integración.* s.l. : Sun Microsystems, 2006.

18. **Dewayne, E y Alexander, Wolf L.** *"Foundations for the study of software architecture".* s.l. : ACM SIGSOFT Software Engineering Notes, 1992.

19. **Garlan, David y Shaw, Mary.** *An introduction to software architecture.* s.l. : CMU Software Engineering Institute Technical Report.

20. **Barbacci, M, y otros.** *Quality Attributes. Carnegie Mellon.* s.l. : University Technical Report., 1995.

21. **Kazman, R, Clements, P y Klein, M.** *Evaluating Software Architectures. Methods and case studies.* s.l. : Addison Wesley. , 2001.

22. **Booch, G, Rumbaugh, J y Jacobson, I.** *The UML Modeling Language User Guide.* s.l. : Adisson Wesley, 1999.

23. **Bass, L, Clements, P y Kazman, R.** *Software Architecture in practice.* s.l. : Addison Wesley, 1998.

24. **Pressman, R.** *Ingeniería de Software. Un Enfoque Práctico Quinta Edición.* s.l. : Mc Graw Hill.

25. **P, Kruchten B.** *The 4+1 View Model of Architecture.* s.l. : IEEE Software, 1995.

26. **Breu, R.** *Towards a Formalization of the Unified Modeling Language.* 1997.

27. *Meta-modelling semantics of UML.* **Evans, A. S y Kent, S.** 1999.

28. *Formal approaches to systems analysis using UML.* **Whittle, J.** 2000.

29. *Formally Modeling UML and its Evolution.* **Toval, A y Fernández-Alemán, J L.** s.l. : Kluwer Academic Publishers, 2000.

30. *Formulations and Formalisms in Software Architecture.* **Shaw, M y Garlan, D.** s.l. : Computer Science Today, 1995.

31. *Assessing the Suitability of a Standard Design Method for Modeling Software Architectures. .* **Medvidovic, N y Rosenblum, D S.** s.l. : Kluwer Academic Publishers., 1999.

32. *Describing Software Architectures with UML.* **Hofmeister, C, Nord, R y Soni, D.** s.l. : Kluwer Academic Publishers, 1999.

33. **Glick, Jeremy.** *The relationship between forward and backward dynamic programming.* 2000.

34. *Comments on Design Patterns for Embedded and Real-Time Systems.* **Tempelmeier, Theodor.** 2001.

35. *Suitability of the UML as an Architecture Description Language with applications to testing.* **Abdurazik, Aynur.** George Mason University : s.n., 2000.

36. *Describing Software Architecture with UML.* **Christine, Hofmeister, Robert, Nord y Dilip, Son.** San Antonio : IEEE Computer Society Press, 1999.

37. **Perera, Jose Raul.** *"ARQUITECTURA DE SOFTWARE PARA SISTEMA GESTION DE INVENTARIOS.* Ciudad de la Habana : s.n., 2007.

38. **Gómez, S. P.** *Swing Univ. Politécnica de Madrid.* 2006.

39. **Bauer, C.** *Hibernate in Action. A Guide to the concepts and practices of object .* s.l. : M. Publications.Co., 2005.

40. **Autores, C. d.** *Guía de Iniciación al Lenguaje JAVA. .* s.l. : Junta de Castilla y León., 1999.

41. [En línea] http://www.dc.uba.ar/materias/arq-soft/2007/cuat1/descargas/ADLs_Estaticos_1.pdf.

42. [En línea] http://triana.escet.urjc.es/aspf/Tema4-1.pdf.

43. [En línea] http://www.sophia.javeriana.edu.co/~cbustaca/Arquitectura%20Software/Presentaciones/arquitecturas_software02.pdf.

44. [En línea] http://www.aprendeenlinea.udea.edu.co/lms/moodle/file.php/120/PropuestasDeArquitectura.pdf .

45. [En línea] http://archjava.fluid.cs.cmu.edu/software/index.html.

46. [En línea] http://www.cs.cmu.edu/~acme/AcmeStudio/.

47. *Klaus-Dieter Schewe. "UML: A modern dinosaur? – A critical analysis of the Unified Modelling Language". En H. Kangassalo, H. Jaakkola y E. Kawaguchi (eds.), Proceedings of the 10th European-Japanese Conference on Information Modelling and Knowledge Bases,.*

Anexos.

Glosario de Términos.

1. **IEEE (Institute of Electrical and Electronics Engineers):** Asociación técnico-profesional mundial dedicada a la estandarización. Es la mayor asociación internacional sin fines de lucro formada por profesionales de las nuevas tecnologías, como ingenieros eléctricos, ingenieros en electrónica, científicos de la computación e ingenieros en telecomunicación.
2. **RUP (Racional Unified Process):** El Proceso Unificado de Desarrollo Software o simplemente Proceso Unificado es un marco de desarrollo software iterativo e incremental. El refinamiento más conocido y documentado del Proceso Unificado es el Proceso Unificado de Rational o simplemente RUP.
3. **UML (Unified Modeling Language):** Es un lenguaje gráfico para visualizar, especificar y documentar cada una de las partes que comprende el desarrollo de software. Entrega una forma de modelar cosas conceptuales como lo son procesos del negocio y funciones de sistema, además de cosas concretas como lo son escribir clases en un lenguaje determinado, esquemas de base de datos y componentes de software reusables.
4. **Abstracción/Enajenación:** Se refiere al énfasis en el "¿qué hace?" más que en el "¿cómo lo hace?", es decir representa el nivel de desconocimiento que tenga una funcionalidad de su implementación. El común denominador en la evolución de los lenguajes de programación, desde los clásicos o imperativos hasta los orientados a objetos, ha sido el nivel de abstracción del que cada uno de ellos hace uso.