UNIVERSIDAD DE LAS CIENCIAS INFORMATICAS

FACULTAD 3

“ARQUITECTURA DE SOFTWARE PARA SISTEMA GESTION DE INVENTARIOS”

TRABAJO PARA OPTAR POR EL TÍTULO DE INGENIERÍA EN CIENCIAS INFORMÁTICAS

AUTOR

Jose Raúl Perera Morales

TUTOR

Ing. Diosmani Meriño Hernández

Ciudad de la Habana

Mayo 2007

DECLARACIÓN DE AUTORÍA

Declaro que soy el único autor de este trabajo y autorizo a la Universidad de las Ciencias Informáticas a hacer uso del mismo en su beneficio.

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

José Raúl Perera Morales

_____

Diosmani Meriño Hechavarría

_____

DEDICATORIA

En primer lugar a la Revolución y a Fidel, por permitirme ser partícipe de su idea.

A mis padres.

AGRADECIMIENTOS

A mis padres por estar a mi lado siempre.

A mi hermano.

A mis amigos de la universidad por apoyarme.

A Grethel Liz por ser mi alma, fuente de inspiración, e impulso cuando ya no había fuerzas.

A Yoan Arlet por tu importante ayuda.

A mi tutor y amigo.

RESUMEN

El presente trabajo de diploma contiene un estudio de los principales elementos que constituyen la Arquitectura de Software, durante su desarrollo se realiza un análisis de dichos elementos con el objetivo de lograr una organización estructural del Sistema de Gestión de Inventario y Almacén, expresada en la relación entre sus componentes, conectores, y restricciones.

El éxito de la aplicación depende en gran medida de las decisiones arquitectónicas tomadas durante su desarrollo, definiendo las características fundamentales de las tecnologías y aspectos esenciales de diseño y proveyendo a los miembros del equipo de una idea clara y objetiva de lo que se está desarrollando.

La Arquitectura para el Sistema de Gestión de Inventarios y Almacén debe de ser modular, flexible, que satisfaga con los requisitos (Analistas), que pueda ser construible (diseñadores y programadores), ser testeable (Calidad) y que cumpla con los parámetros de funcionalidad, eficiencia y confiabilidad.

Palabras Claves

- **Arquitectura de software**: Es la organización fundamental de un sistema encarnada en sus componentes, las relaciones entre ellos, el ambiente y los principios que orientan su diseño y evolución.

ÍNDICE

INTRODUCCIÓN

Los cambios tecnológicos en el campo de la informática hacen que esta ciencia sea cada día más dinámica. Tecnologías como las redes, el comercio electrónico, las de tele trabajo, entre otras están teniendo tal repercusión en el mundo empresarial cubano, que cada vez resulta más difícil sustraerse a esta corriente de innovación.

La economía cubana a partir de la desintegración del campo socialista de Europa del Este, la URSS y el arreciamiento del bloqueo por parte de los gobiernos estadounidenses, provocó que la introducción de mejoras tecnológicas en el país se vieran frenadas y que las que estaban en uso se volvieran caducas. Nuestro país en aras de revertir esta situación ha invertido grandes recursos en la medida que la economía lo permitiera.

De este modo, durante los primeros años de este siglo, el país se ha centrado en un proceso de informatización de la sociedad, lo que trajo consigo la necesidad de crear sistemas eficientes y a la altura de las nuevas tecnologías, que como parte del los avances económicos que va teniendo el país, se van introduciendo.

A partir del proceso de perfeccionamiento empresarial que se lleva en el país, las empresas trabajan para lograr una mayor eficiencia en su producción o en los servicios que brindan; como parte de este perfeccionamiento está el control de los inventarios y existencias de activos. La introducción de un sistema capaz de automatizar esta actividad se hace una necesidad, ante la posibilidad de desvíos de recursos y frente a la lucha contra la corrupción, además de permitir mejorar la dinámica de las empresas y la mejora de us procesos .

Existen en el país diversos sistemas de inventario y almacén, estos sistemas, debido a los cambios que se gestan en la economía, se han vuelto caducos al no recoger los nuevos requisitos de los clientes y no ser flexibles, por ejemplo el IHMM 2000 de la empresa GET (Grupo Electrónico del Turismo). La tecnología utilizada en dichos sistemas es atrasada y no se corresponde con los nuevos requisitos de interfaz y de seguridad, además de que no incluye todas las funcionalidades que pudieran automatizarse en aras de mejorar los procesos de control.

De aquí surge la propuesta de proyecto de realizar un sistema de inventario y almacén que se modele como un sistema estándar y parametrizable, capaz de adaptarse a las necesidades particulares de los clientes.

¿Cómo lograr que los desarrolladores de este sistema tengan una idea clara de lo que están desarrollando?

La arquitectura de software involucra los elementos más significativos del sistema y está influenciada entre otros por las plataformas software, sistemas operativos, manejadores de bases de datos, protocolos, consideraciones de desarrollo, como sistemas heredados y requerimientos no funcionales. Es una radiografía del sistema que se está desarrollando, lo suficientemente completa como para que todos los implicados en el desarrollo tengan una idea clara de qué es lo que están construyendo, pero lo suficientemente simple como para que si se suprime algo, una parte importante del sistema quede sin especificar.

De una forma solapada y sin apenas darse cuenta los desarrolladores de software han ido desarrollando la arquitectura de software. Por mera lógica buscaron la forma de reutilizar funciones o heredarlas de sistemas anteriores, la utilización y búsqueda de solución a problemas comunes como: estrategias de organización de los sistemas, sería la primera utilización de patrones que se conoce, aunque no se escribieran de la forma en la que hoy los conocemos.

La Arquitectura de Software, se definió como disciplina de software a partir del año 1992 en la publicación “Foundations for the study of software architecture” escrito por Dewayne Perry, Alexander Wolf donde planteaban [Perry, Wolf, 1992]:

“*La década de 1990, creemos, será la década de la arquitectura de software…*”

No es sino hasta el año 2000 que la IEEE hace una definición de Arquitectura de Software (IEEE 1471-2000) y que es aceptada por la gran mayoría de los arquitectos y desarrolladores.

Desde entonces, las descripciones arquitectónicas han sido enfocadas directamente hacia ciertas expresiones como arquitectura en capas o cliente/servidor, etc. Estas limitan el sentido de la arquitectura de software como disciplina que es, y su utilidad en el proceso de desarrollo.

Es evidente la necesidad de la representación arquitectónica en los sistemas de software, en primer lugar las representaciones arquitectónicas elevan el nivel de abstracción, facilitando la comprensión de sistemas complejos. En segundo lugar hace que aumente la posibilidad de reutilizar los distintos componentes o elementos de software y que se pueda reutilizar hasta la misma arquitectura.

Lo antes expuesto llevó a definir el siguiente **problema**:

¿Cómo tomar decisiones significativas sobre la organización del Sistema Gestión de Inventario y Almacén y la selección de elementos estructurales sobre los cuales se constituye el mismo, su comportamiento y la colaboración entre dichos elementos?

Para enfocar la investigación se planteó **objetivo**:

Definir una arquitectura de software para el Sistema Gestión de Inventario y Almacén, que permita al arquitecto tomar decisiones significativas sobre la organización del sistema, la selección de elementos estructurales sobre los cuales se constituye el sistema, su comportamiento y la colaboración entre dichos elementos y que permita a los desarrolladores tener una idea clara de lo que se está desarrollando.

Objeto de estudio:

Proceso de desarrollo del Sistema de Gestión de Inventario y Almacén.

Dentro del objeto de estudio el **campo de acción** en el que centrará la investigación será:

Diseño y descripción arquitectónica para el Sistema de Gestión de Inventario y Almacén.

La investigación se desarrolló a través de las siguientes **tareas**:

1. Revisar la bibliografía existente para establecer una comparación entre los distintos enfoques de la arquitectura y ver cómo han evolucionado.
2. Revisar bibliografía científica teórica usada en el desarrollo de arquitecturas para sistemas empresariales desarrollados anteriormente para estimar el grado de novedad de los posibles resultados.

3. Identificar patrones arquitectónicos a utilizar, fundamentación de estos patrones, así como su aplicación.
4. Definir las herramientas a utilizar para el desarrollo.
5. Definir la estrategia de desarrollo.

Para realizar las tareas se emplearon los siguientes **métodos**:

Métodos teóricos:

- Análisis y Síntesis: para el procesamiento de la información y arribar a las conclusiones de la investigación, así como para precisar las características del modelo arquitectónico propuesto.
- Histórico - Lógico: Para determinar las tendencias actuales de desarrollo de los modelos y enfoques arquitectónicos.
- Inducción y deducción: A partir del estudio de distintos estilos y modelos de arquitecturas arribar a proposiciones de estilos de arquitecturas específicas.
- Método Sistémico: Para determinar los componentes y definir las relaciones entre estos.

Métodos empíricos:

- Observación: Para la percepción selectiva de las restricciones y propiedades del sistema, y sistémica para el control de la evolución de la arquitectura inicial.

Los **aportes prácticos esperados del trabajo**:

Se espera obtener una arquitectura para el Sistema de Gestión de Inventarios y Almacén, modular, flexible, que satisfaga con los requisitos (Analistas), que pueda ser construible (diseñadores y programadores), ser testeable (Calidad). Qué cumpla con los parámetros de funcionalidad, eficiencia y confiabilidad.

Este trabajo está estructurado en tres capítulos:

Capítulo 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.

Capítulo 2: Diseño arquitectónico:

En este capítulo se hace una propuesta de solución al problema panteado en el diseño teórico de la investigación.

La solución propuesta está dividida en dos subtópicos principales:

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

Se presenta además una valoración personal del autor del trabajo sobre la propuesta de desarrollo presentado.

CAPITULO 1

Fundamentación teórica

1.1. Introducción

En este capítulo se realiza un análisis del estado del arte de la arquitectura de software, sobre los distintos enfoques y tendencias de la arquitectura de software para sistemas empresariales y de gestión, que permitan modelar el marco teórico y el modelo conceptual sobre el que se fundamenta la investigación.

1.2. Ingeniería de Software 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 [Trujillo, 2006] y la concepción de que es mucho más fácil y rápido reutilizar componentes ya desarrollados y probados, a tener que implementarlos desde cero.

Los antecedentes más significativos de las plataformas de componentes se pueden establecer en DCE y CORBA, desarrolladas por iniciativa de los consorcios OSF (Open Software Foundation) y OMG (Object Management Group), respectivamente. Aunque DCE (Distributed Computing Environment) [OSF, 1994] tuvo en un principio una buena acogida por parte de la industria, algunas de las limitaciones que presenta (no es orientada a objetos, no define servicios de transacciones y solo permite invocaciones estáticas, no dinámicas) le han hecho perder gran parte de su cuota de mercado en el desarrollo de aplicaciones distribuidas. Sin embargo, la especificación de la arquitectura CORBA (Common Object Request Broker Architecture) [OMG, 1999, Vinoski, 1998], 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, COM (Component Object Model), desarrollada

por Microsoft [Rogerson, 1997, Box, 1998] y JavaBeans, desarrollada por Sun [Sun Microsystems, 1997].

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 CCM (CORBA Component Model), EJB (Enterprise Java Beans), DCOM (Distributed COM) [Microsoft, 1996].

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) [Fayad et al., 1999]. 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.

Otra forma de expresar este concepto es que un marco de trabajo es el esqueleto de una aplicación que debe ser adaptado por el programador según sus necesidades concretas. [Johnson, 1997].

El desarrollo de aplicaciones a partir de un framework se lleva a cabo mediante la extensión del mismo, para lo cual el usuario debe tener información acerca de cuáles son sus puntos de entrada y cómo adaptarlos.

Un framework se considera como la plantilla (template) 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 nos 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.

Los componentes no son separables de la arquitectura. Así mismo, todos los modelos y plataformas de componentes imponen una serie de compromisos sobre dichos componentes, indicando la forma en que se describe su interfaz o que mecanismos básicos tienen que proporcionar [Brown y Wallnau, 1998]. La interfaz de un componente no solo muestra su funcionalidad, de forma abstracta, sino que implica además toda una serie de restricciones arquitectónicas ligadas a la plataforma o modelo del que forma parte. [Szyperski, 2000].

¿Cómo influirá la arquitectura en el desarrollo del Sistema de Gestión de Inventarios?

La arquitectura de software basada en componentes permitiría utilizar componentes ya probados que no tuvieran errores y que acortaría el tiempo de desarrollo, permitiría además la creación de nuestros propios componentes que deberán ser documentados y reutilizables por otros subsistemas de la aplicación u otros sistemas.

De aquí, que la arquitectura al estar basada en componentes proporcione la herramienta fundamental para lograr hacer una Ingeniería Basada en Componentes, que cumpla con las expectativas de reusabilidad y reutilización de las partes de software.

1.3. Arquitectura de Software

A medida que crece la complejidad de las aplicaciones, y que se extiende el uso de sistemas distribuidos y sistemas basados en componentes, los aspectos arquitectónicos del desarrollo de software están recibiendo un interés cada vez mayor, tanto desde la comunidad científica como desde la propia industria del software.

La arquitectura de software es una disciplina reciente [Szyperski, 2000]. 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"[Shaw, Garlan, 1996].

Esta definición es la base de la corriente estructuralista de la arquitectura representada por Mary Shaw y David Garlan de la escuela de Carnegie Mellon, esta tiene variantes con modelos de datos (Medvidovic), radicales y formales (Moriconi - SRI).

Esta definición no es capaz de definir el momento en el ciclo de desarrollo del software donde se debe llevar a cabo la arquitectura, según esta, el desarrollo de la misma comienza en un momento entre la definición de los requisitos y la modelación del sistema o entre la modelación del sistema, el análisis y el diseño.

Rational Unified Process, 1999, propone que:

"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 esos 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" [Jacobson, et al, 1999].

Esta define la arquitectura como una etapa más de la Ingeniería de Software enfocada a la orientación a objetos y a UML, como lo plantea la metodología de desarrollo RUP, representa la arquitectura como algo más que la simple composición de herramientas o tecnologías a usar en el desarrollo del sistema.

Existen otras corrientes de arquitecturas como:

- Basada en patrones, que se basa principalmente en la redefinición de los estilos como patrones POSA [Wellicki, 2007], un ejemplo significativo de esta corriente es Microsoft Pattern & Practices.[Microsoft MSDN, 2000].
  En la corriente de arquitectura de Microsoft utiliza los patrones POSA [Wellicki, 2007] como estilos de arquitectura, ejemplificados en sus frameworks de desarrollo (MSF).
- Arquitecturas procedural y metodológicas, sus principales exponentes son: Kazman y Bass (SEI), que proponen variantes de arquitecturas basadas en escenarios.[Microsoft MSDN, 2006]

La IEEE 1471 - 2000 definió lo que es Arquitectura de Software como: la organización fundamental de un sistema encarnada en sus componentes, las relaciones entre ellos, el ambiente y los principios que orientan su diseño y evolución.

Cuando se habla de componentes es importante establecer una diferencia entre componentes de software y los componentes de la arquitectura, por lo que los arquitectos en las definiciones modernas prefieren llamarle "*elementos*".

Los componentes de la arquitectura o "elementos" pueden ser dinámicos o estáticos, estos son elementos ya sea de procesamiento, de datos o de conexión [Reynoso, 2004] mientras que los componentes de software se refiere a subsistemas, clases, interfaces, etc.

Esta definición dada por la IEEE, define la arquitectura no como un flujo de trabajo dentro de una metodología, deja claro que la Arquitectura de Software 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 y constituye la organización del sistema al nivel más alto de abstracción.

Entonces podemos concluir que:

La Arquitectura de Software no es:

- Una disciplina madura.
- Diseño de software.
- Vinculada a la metodología RUP o ninguna otra metodología.

Es importante resaltar que la arquitectura sí tiene una relación directa con los requerimientos no funcionales de un sistema, los que permiten definir escenarios para la descripción arquitectónica. Los requerimientos funcionales se satisfacen mediante modelado y diseño de la aplicación.

Entonces la arquitectura es:

- Una vista estructural de alto nivel.
- Define estilos o combinación de estilos para dar una solución.

- Se concentra en los requerimientos no funcionales.

1.4. Patrones de Arquitectura

En las secciones anteriores hemos caracterizado la arquitectura de software como la disciplina que aborda los aspectos estructurales de la aplicación desde un diseño de alto nivel.

¿Qué sucede cuando existen problemas comunes en esta etapa de diseño?

Entendemos por patrón una solución probada que se puede aplicar con éxito a un determinado tipo de problemas que aparecen repetidamente en algún campo. El establecimiento de estos patrones comunes es lo que posibilita el aprovechamiento de la experiencia acumulada en el diseño de aplicaciones.

Un patrón codifica conocimientos específicos acumulados por la experiencia en un dominio, por tanto podemos decir que, un sistema bien estructurado está lleno de patrones. Cada patrón describe un problema que ocurre una y otra vez en nuestro ambiente y luego describe el núcleo de la solución a ese problema, de tal manera que puede usarse esa misma solución tantas veces como ocurra ese problema.

Los elementos de un patrón son:

1. Nombre
    a. Define un vocabulario de diseño.
    b. Facilita la abstracción.
2. Problema
    a. Describe cuando aplicar el patrón
    b. Conjunto de fuerzas: objetivas y restricciones.
    c. Prerrequisitos.
3. Solución
    a. Elementos que constituyen el diseño (plantilla).
    b. Forma canónica para resolver fuerzas.
4. Consecuencias
    a. Resultados

b. Extensiones
c. Consensos

Al contrario de los estilos arquitectónicos los patrones son muchos y muy variados y es casi imposible revisar todos los patrones que existen a la hora de hacer una determinada aplicación, por eso se recomienda el uso de los patrones que estén predeterminados en cada uno de los estilos que se seleccionen para la arquitectura.

Los siguientes son algunos de los patrones más usados, divididos en las principales categorías de patrones arquitectónicos [Gamma, 2002]:

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

    Organiza lógica del negocio por los procedimientos, donde cada procedimiento maneja una sola petición de la presentación.

    Este patrón solo se utiliza en aplicaciones poco complejas por tanto no se recomienda su uso en este sistema.
  - Domain Model (Modelo de dominio)

    Modelo del objeto del dominio que incorpora comportamiento y datos.
  - Service Layer (Capa de servicio)

    Define el límite de un uso con una capa de servicios que establezca 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.
- Data Source Architectural Patterns (Patrones Arquitectónicos de Fuente de Datos)
  - Active Record (Registro Activo)

    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 eso los datos
  - Data Mapper (Mapeo de Datos)

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

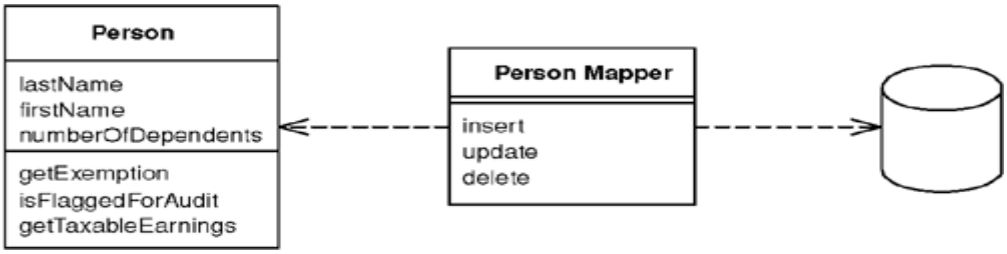


Fig. 1. Capa de mapeo [Palos, 2006]

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

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

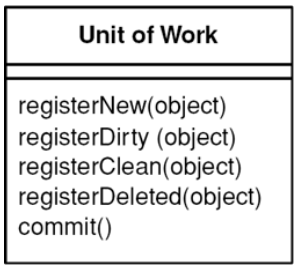


Fig. 2. Clase unidad de trabajo [Palos, 2006]

  - Identity Map (Mapa de Identificación)

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

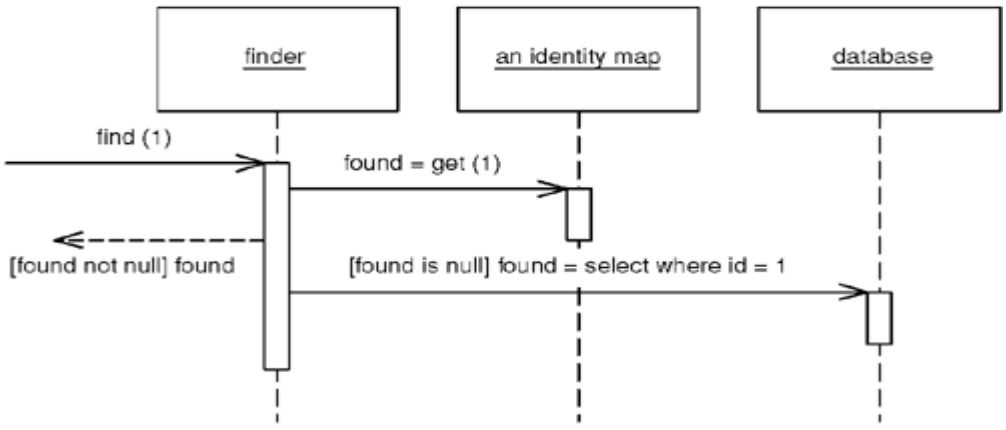


Fig. 3. Patrón mapa de identificación [Palos, 2006]

- Lazy Load (Carga Tardía)

  Un objeto que no contiene todos los datos que necesitas pero sabe conseguirlo

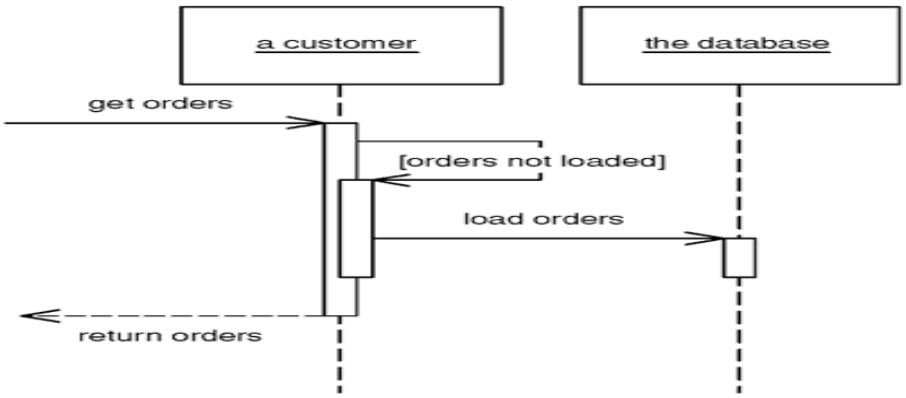


Fig. 4. Patrón carga tardía [Palos, 2006]

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

    Mapea las relaciones en dependencias de las llaves foráneas de las tablas.
  - Association Table Mapping (Mapeo de Tablas Asociadas)

    Crea asociaciones entre las tablas asociadas, y los respectivos mapeos de las tablas.
  - Dependent Mapping (Mapeo Herencia)

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

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

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

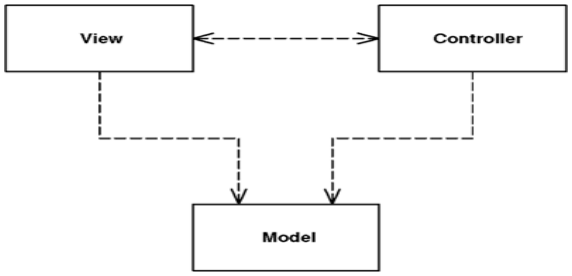


Fig. 5. Patrón Modelo – Vista- Controlador [Palos, 2006]

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

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

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

    Almacena el estado de la sesión en el cliente

- J2EE Patterns [Sun Microsystems, 2002]
  - 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.

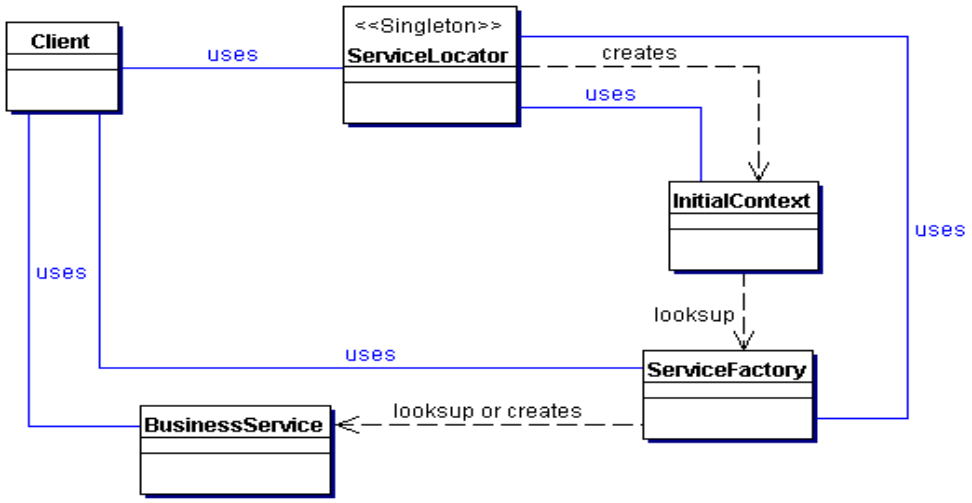


Fig. 6. Patrón service locator [Palos, 2006]

- Data Access Object (Objetos de Acceso a Datos)

  El acceso a los datos varía dependiendo de la fuente de los datos. Tener acceso al almacenaje persistente, por ejemplo a una base de datos, varía grandemente dependiendo del tipo de almacenaje (bases de datos orientadas al objeto, ficheros " planos")

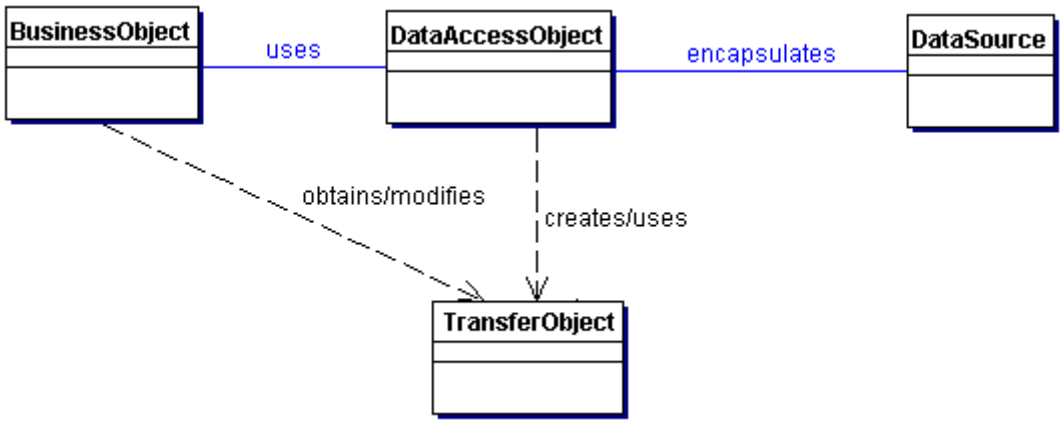


Fig.7. Patrón Objeto de Acceso a Datos [Palos, 2006]

1.5. Niveles de abstracción arquitectónicos

La Arquitectura del Software se encarga de la descripción y estudio de las propiedades estructurales de los sistemas de software. Sin embargo, el estudio de dichas propiedades puede llevarse a cabo desde diferentes niveles de abstracción.

1.5.1. Estilos arquitectónicos

El éxito en el trabajo arquitectónico radica en la elección del estilo arquitectónico.

¿Qué es un estilo arquitectónico?

Cuando se habla del término de estilo arquitectónico en la arquitectura de software no queda más remedio que establecer una comparación entre estilos arquitectónicos de la arquitectura civil, por ejemplo: estilos eclécticos, barrocos, clásicos, etc. Los estilos de la arquitectura de

software definen un conjunto de concepciones arquitectónicas comunes que identifican un momento en el desarrollo de la arquitectura.

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 4 C [Perry, Wolf, 1992]:

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

Estos permiten sintetizar estructuras de soluciones que luego serán refinadas a través del diseño, son pocos los estilos (alrededor de 20) que encapsulan una enorme cantidad de configuraciones concretas. Definen los patrones posibles de las aplicaciones, evitando errores arquitectónicos y permiten evaluar arquitecturas alternativas con ventajas y desventajas conocidas ante diferentes conjuntos de requerimientos no funcionales.

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 por ejemplo: patrón Layers (Capas), Tuberías y filtros, entre otros; los tipos de componentes presentes habitualmente en el estilo y las interacciones que se establecen entre ellos.

Algunos de los principales estilos arquitectónicos se usan en la actualidad están divididos por Clases de Estilos las que engloban una serie de estilos arquitectónicos específicos [Shaw y Garlan, 1996, Bass et al., 1998]:

- Estilos de Flujo de datos
    - Tuberías y Filtros
- Estilos centrados en datos
    - Arquitecturas de Pizarra o repositorio

- Estilos de Llamada y Retorno
    - Modelo – Vista – Controlador (MVC)
    - Arquitectura en Capas
    - Arquitectura Orientada a Objetos
    - Arquitectura basada en Componentes
- Estilo de Código Móvil
    - Arquitectura de Máquinas Virtuales
- Estilos Peer – To – Peer
    - Arquitectura basadas en Eventos
    - Arquitecturas Orientas a Servicios (SOA)

Además de determinar los tipos de componentes y conectores involucrados, un estilo arquitectónico indicaría cuáles son los patrones y restricciones de interconexión o composición entre ellos, lo que denomina como invariantes del estilo. Por último, asociadas a cada estilo hay una serie de propiedades que lo caracterizan, determinando sus ventajas e inconvenientes, y que condicionan la elección de uno u otro estilo para resolver un determinado problema. [Canal, 2000]

Las especificaciones de los estilos más importantes a analizar para su utilización en la organización del sistema Gestión de Inventario y Almacén son:

- **Modelo – Vista – Controlador**

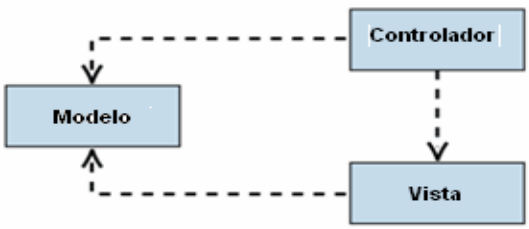


Fig. 8. Estilo Modelo-Vista-Controlador

Se utiliza principalmente cuando es necesario modularizar la interfaz de usuario, las reglas de negocio y el control de eventos.

Modelo: Administra el comportamiento y los datos del dominio de aplicación, responde a requerimientos de información sobre su estado (usualmente formulados desde la vista y responde a instrucciones de cambiar el estado habitualmente desde el controlador). Mantiene el conocimiento del sistema. No depende de ninguna vista y de ningún controlador.

Vista: Maneja la visualización de la información.

Controlador: Interpreta las acciones del ratón y el teclado, informando al modelo y/o a la vista para que cambien según resulte apropiado.
Tiene tres variantes principales: Activa, Pasiva y Documento-Vista

Ventajas

- Se puede mostrar distintas variantes de interfaz gráfica simultáneamente.
- La interfaz tiende a cambiar más rápido que las reglas del negocio. Agregar nuevos tipos de vista no afecta el modelo.
- Evita poner el código indebido en la capa impropia. Facilita despliegue en caso de modificaciones en el modelo de datos.

Desventajas

- Puede aumentar un poco la complejidad de la solución. Como está guiado por eventos puede ser algo más difícil de depurar.
- Si hay demasiados cambios en el modelo la vista puede provocar un constante refrescamiento de las vistas, a menos que se prevea programáticamente.

- **Arquitectura basada en Objetos**

Pertenece al estilo de Llamada y Retorno. 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, los objetos representan una clase de componentes que ellos llaman *managers*, debido a que son responsables de preservar la integridad de su propia representación.

Los componentes del estilo se basan en principios Orientado a Objetos: encapsulamiento, herencia y polimorfismo. Son así mismo las unidades de modelado, diseño e implementación, y

los objetos y sus interacciones son el centro de las incumbencias en el diseño de la arquitectura y en la estructura de la aplicación.

Las clases interfaces están separadas de sus implementaciones. En general la distribución de objetos es transparente. El mejor ejemplo de Orientación a Objetos para sistemas distribuidos es Common Object Request Broker Architecture (CORBA), en la cual las interfaces se definen mediante Lenguaje de Descripción de Interfaces (Interface Description Language, IDL); un Object Request Broker media las interacciones entre objetos clientes y objetos servidores en ambientes distribuidos.

En tanto componentes, los objetos interactúan a través de invocaciones de métodos y mensajes. Hay muchas variantes del estilo; algunos sistemas, por ejemplo, admiten que los objetos sean tareas concurrentes; otros permiten que los objetos posean múltiples interfaces facilitando distinto comportamiento.

Ventajas

- Refinamiento (descomposición funcional) es sencillo.
- Encapsulamiento reutilización, fácil descomposición en una colección de agentes interactivos.

Desventajas

- Propagación de los cambios por ejemplo: nombres y tipos de argumentos, etc.
- Dependencia en cascada.
- Si se cambia un objeto se deben modificar todos los objetos o métodos que lo invocan.
- Granularidad muy fina para sistemas grandes.

- **Arquitectura en Capas**

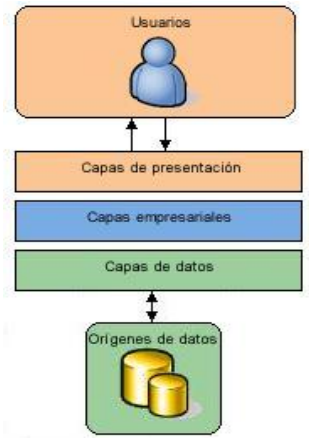


Fig. 9. Estilo Capas

Resulta útil cuando se pueden identificar distintas clases de servicios que pueden ser articulados jerárquicamente. Los componentes de cada capa consisten en conjuntos de clases. Las interacciones entre las capas ocurren generalmente por invocación de métodos, por definición los niveles de abajo no deben poder utilizar funcionalidad ofrecida por los de niveles superiores.

Ventajas

- Modularidad del sistema.
- Facilita la localización de errores.
- Mejora soporte del sistema

Desventajas

- Puede ser difícil definir que componentes ubicar en cada una de las capas.

- **Arquitectura Orientada a Servicios (SOA)**

"La recompensa potencial (de SOA) es enorme para las empresas que entiendan esta evolución y se muevan hacia estas arquitecturas. ... La tecnología de computación distribuida promete ser lo suficientemente flexible y elegante para responder a las necesidades de negocios y proporcionar la agilidad de negocios que las compañías han anhelado tanto tiempo, pero siempre ha estado fuera de alcance". [The Rational Edge, 2004].

Gartner:

“SOA es una arquitectura de software que comienza con una definición de interfaz y construye toda la topología de la aplicación como una topología de interfaces, implementaciones y llamados a interfaces. Sería mejor llamada “arquitectura orientada a interfaces”. SOA es una relación de servicios y consumidores de servicios, ambos suficientemente amplios para representar una función de negocios completa”.

	Programación Estructurada	Objetos	Componentes	Servicios
Granularidad	Muy fina	Fina	Intermedia	Gruesa
Contrato	Definido	Privado/Público	Público	Publicado
Reusabilidad	Baja	Baja	Intermedia	Alta
Acoplamiento	Fuerte	Fuerte	Débil	Muy débil
Dependencias	Tiempo de Compilación	Tiempo de Compilación	Tiempo de Compilación	Ejecución
Ámbito de Comunicación	Intra-Aplicación	Intra-Aplicación	Intra-Aplicación	Inter-Empresas

Tabla 1. Propiedades de la arquitectura SOA.

La elección del estilo arquitectónico está dado por los escenarios que se puedan identificar de los requerimientos no funcionales. Se puede crear combinaciones de estilos arquitectónicos permitiendo lograr la satisfacción de los distintos requerimientos o escenarios.

1.5.2. Modelos y arquitecturas de referencia

Los modelos o arquitecturas de referencia especifican un estilo de arquitectura y lo particularizan realizando una descomposición y definición estándar de componentes, a los que les impone una serie de restricciones para su organización.

Existen muchos modelos de referencia. Quizás el más conocido de todos ellos sea Interconexión de Sistemas Abiertos (Open Systems Interconnection, OSI) [CCITT/ITU-T, 1988], recomendado por la ISO. Se trata de una arquitectura de referencia que particulariza el estilo arquitectónico de organización en capas, estableciendo en siete el número de niveles y definiendo cuáles son las funciones que debe realizar cada nivel y cuál ha de ser su interfaz.

Otro ejemplo es el modelo de referencia RM-ODP para el diseño de sistemas distribuidos y abiertos [ISO/ITU-T, 1995], que tiene por objeto conseguir que la heterogeneidad de las plataformas de hardware, sistemas operativos, lenguajes, etc. sea transparente al usuario.

Este modelo, basado en el estilo llamado de componentes independientes, típico de las aplicaciones distribuidas, contiene tanto elementos descriptivos como prescriptivos. Los primeros definen de forma precisa el vocabulario y conceptos que aparecen en el desarrollo de un sistema distribuido de cualquier tipo. Por su parte, los elementos prescriptivos imponen una serie de restricciones que deben cumplir los sistemas distribuidos para formar parte de este modelo.

1.5.3. Marcos de Trabajo (Frameworks)

El Marco de trabajo o Framework en su término en inglés, define una arquitectura adaptada a las particularidades de un determinado dominio de aplicación, definiendo de forma abstracta una serie de componentes y sus interfaces, y estableciendo las reglas y mecanismos de interacción entre ellos.

Normalmente se incluye además la implementación de algunos de los componentes o incluso varias implementaciones alternativas. Tal como hemos visto, la labor del desarrollador será: seleccionar, instanciar, extender y reutilizar los componentes que éste proporciona y completar la arquitectura del mismo desarrollando componentes específicos, que deben ser encajados en el framework, logrando así desarrollar diferentes aplicaciones siguiendo las restricciones estructurales impuestas por el framework. [Carlos Canal, 2000]

1.5.4. Familias y líneas de productos

Las familias o líneas de productos son un conjunto de aplicaciones similares, o bien diferentes versiones o configuraciones de la misma aplicación, de forma que todas estas aplicaciones tienen la misma arquitectura, pero cada una de ellas está adaptada o particularizada al entorno o configuración donde se va a ejecutar. Además de compartir una arquitectura, en los

productos de la línea se reutilizan toda una serie de componentes que son, precisamente, los que caracterizan la línea de productos [Bosch, 2000]

Como se puede observar el establecimiento de líneas o familias de productos está muy ligado a la Gestión de Configuración de Software, que define las bibliotecas de componentes y regulan el control de versiones.

Las ventajas de establecer estas líneas de productos tienen que ver fundamentalmente con la reutilización, ya que implican una disminución de costes y un aumento de la calidad y fiabilidad del producto. Según [Canal, 2000] estas ventajas se materializan en:

- **Reutilización de la arquitectura**. Todas las aplicaciones de la línea comparten básicamente la misma arquitectura, lo que permite reutilizar su diseño, su documentación, asegurar el cumplimiento de determinadas propiedades.
- **Reutilización de los componentes**. Como ya se ha indicado, parte de los componentes son comunes a los distintos productos, o bien pueden obtenerse parametrizando componentes genéricos. Esto facilita notablemente la implementación.
- **Fiabilidad.** Los componentes empleados están exhaustivamente probados, al ser compartidos con aplicaciones que ya están en funcionamiento, lo que permite corregir defectos y aumentar su fiabilidad y calidad. Por otro lado, el uso de dichos componentes en aplicaciones que comparten la misma arquitectura permite garantizar que no haya pérdida de fiabilidad en la reutilización.
- **Planificación**. La experiencia repetida en el desarrollo de productos de la línea permite hacer una planificación más fiable del proyecto de desarrollo, y con mayores posibilidades de ser cumplida.

1.5.5. Instancias arquitectónicas

Las instancias arquitectónicas, representan la arquitectura de un sistema de software concreto, caracterizada por los módulos o componentes que lo forman y las relaciones que se establecen entre ellos. Como es lógico, cualquier aplicación tiene una arquitectura que puede ser descrita y estudiada y que presenta determinadas propiedades, ya sea o no la aplicación parte de una familia o línea de productos, haya sido o no desarrollada utilizando un framework, siga o no un

modelo o arquitectura de referencia, o incluso pueda ser catalogada dentro de un estilo arquitectónico o siga un estilo heterogéneo. [Canal, 2000]

1.6. Diseño arquitectónico

La descripción arquitectónica depende de mucho más que la simple descripción de plataformas o frameworks de trabajos, sino que depende de los componentes (elementos), con aserción de propiedades, interfaces e implementaciones, los conectores, las configuraciones o sistemas de abstracción y encapsulamiento, los requisitos no funcionales, las restricciones, los estilos, la evolución y hasta las herramientas de verificación.

Existen varios tipos de lenguajes utilizados para describir la arquitectura de un sistema aquí solo se analizarán dos tipos esenciales.

1.6.1. Lenguajes de Descripción Arquitectónica

Existen herramientas de modelado que soportan desarrollos basados en arquitecturas, estos son los ADLs (Architecture Description Languages).

Estos lenguajes permiten construir una estructura de alto nivel, es decir sin detalles de implementación. Los ADLs son poco utilizados aunque algunos de ellos generan el template de la aplicación, la presencia de UML ha hecho que estos lenguajes no hayan ocupado el lugar que debían.

Entre las características a considerar de estos lenguajes se puede citar las siguientes [Shaw y Garlan, 1996]:

- **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.

Entre los principales ADLs que se usan en la actualidad están:

- ACME
- Armani
- Jacal
- CHAM

1.6.2. Lenguaje Unificado de Modelado

RUP (Proceso Unificado de Software) postula un proceso de desarrollo iterativo, incremental, guiado por los casos de uso y centrado en la arquitectura [Booch et al., 1999] El Lenguaje Unificado de Modelado (UML) es una notación para especificar, visualizar y documentar sistemas de software desde la perspectiva orientado a objetos. Su objetivo es representar el conocimiento acerca de los sistemas que se pretenden construir y las decisiones tomadas durante su desarrollo, tanto los representados por diagramas estáticos (Casos de Uso, diagrama de clases, etc.) como los dinámicos (Diagramas de actividades, interacción, etc.)

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 lógica, 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 [Kruchten, 1995].

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 [Rumbaugh et al., 1999], cuya semántica se define de manera semi-formal por medio de un meta-modelos, textos descriptivos y restricciones. Existen además numerosas iniciativas para dotar a esta notación de una semántica formal pero que aun no se ha consolidado.

Según diversos autores UML no es, en sí, un lenguaje de descripción arquitectónica pues su forma de expresar ciertas características, sobre todo dinámicas de las estructuras no es suficiente para los arquitectos. [Medvidovic y Rosenblum, 1999]

1.7. Proceso Unificado de Desarrollo y Arquitectura de Software

Según el Proceso Unificado de Desarrollo de Software la arquitectura involucra los elementos más significativos del sistema y está influenciada entre otros por plataformas software, sistemas operativos, manejadores de bases de datos, protocolos, consideraciones de desarrollo como sistemas heredados y requerimientos no funcionales. Se representa mediante varias vistas que se centran en aspectos concretos. [Ayuda Rational, 2003]

Todas las vistas juntas forman el llamado modelo 4+1 de la arquitectura, recibe este nombre porque lo forman las vistas lógica, de implementación, proceso y despliegue, más la de casos de uso que es la que da cohesión a todas.

La metodología de desarrollo RUP (Proceso Unificado de Software) plantea que el rol de arquitecto es el responsable por el desarrollo y mantenimiento de la arquitectura de software, la que incluye como hemos visto las principales decisiones técnicas y las restricciones sobre el diseño e implementación del proyecto.

El rol de arquitecto existe en otras metodologías de desarrollo, con actividades diferentes pero con el mismo objetivo dentro del equipo de desarrollo.

¿Qué características debe cumplir un arquitecto?

“El arquitecto ideal debe ser una persona de letras, matemático, al corriente de estudios históricos, un estudiante diligente de la filosofía, conocido de la música, no ignorante de medicina, entendido en las respuestas de consultoría jurídica, al corriente de astronomía y de cálculos astronómicos” [Ayuda Rational, 2003]

Resumiendo, el arquitecto del software debe tener visión, y una profundidad de la experiencia que permite tomar decisiones rápidamente y hacer el juicio adecuado, crítico en ausencia de la información completa [Ayuda Rational, 2003]

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, 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.

En este caso no se propondrá en la solución la utilización de la vista de procesos, ya que esta es opcional dependiendo de las características del sistema.

Los principales artefactos a desarrollar son:

- Documento Descripción de la Arquitectura (4 Vistas)
- Modelo de Despliegue
- Modelo de Implementación
- Protocolos
- Interfaces

1.8. Calidad en la Arquitectura de Software

Los requerimientos no funcionales definen los escenarios, las tácticas a utilizar y los frameworks de desarrollo, todo en función de satisfacer los atributos de:

- Funcionamiento

- Disponibilidad
- Modificabilidad
- Seguridad
- Verificabilidad (testeability)
- Gestionabilidad
- Usabilidad

Hacer un adecuado balance entre la mantenibilidad, la interoperabilidad, la portabilidad, la funcionalidad, la seguridad, la disponibilidad, y la reusabilidad, entre otros atributos, es esencial para obtener un sistema que cumpla las expectativas de las distintas partes interesadas. [Bastarrica, 2005]

El diseño de una arquitectura que tenga en cuenta los atributos deseados, sólo permitirá que el sistema resultante tenga estas características, pero no lo garantiza.

Estos atributos se pueden dividir en dos grandes grupos los que se pueden verificar en la ejecución del software y los que no, dentro del primer grupo se encuentran el funcionamiento, seguridad, disponibilidad, funcionalidad y usabilidad. Dentro del segundo grupo, y no menos importantes, están la modificabilidad, portabilidad, reusabilidad, integrabilidad y verificabilidad.

La solución propuesta debe:

- Satisfacer los requisitos (analistas)
- Ser construible (programadores & diseñadores)
- Ser probada (Administrador de Calidad)
- Ser administrable (administradores)

La descripción de la solución debe:

- Ser evaluable (por otros arquitectos)

El proceso de construcción debe:

- Ser manejable (por el jefe de proyecto)

¿Qué es una solución evaluable?

- Debe permitir evaluar compromisos y opciones.
- Por tanto, debe tener rastreabilidad entre las partes de la solución y del problema.

Proceso manejable:

- Debe ser particionada e indicar dependencias.
- Particiones: unidades naturales de asignación de trabajo.
- Dependencias: la base para definir calendario.

La solución debe satisfacer requisitos

- Debe convencer al analista.

La IEEE std 1061, 1992 plantea como principales atributos de calidad:

- Eficiencia
- Funcionalidad
- Mantenibilidad / Portabilidad
- Confiabilidad
- Usabilidad

Existen métodos para la evaluación de la arquitectura propuesta por ejemplo:

- SAAM

  Este método de evaluación está basado en escenarios y permite evaluar una arquitectura o evaluar y comparar varias.

- ATAM (Architecture Trade Off Analysis Method)

  Este método de evaluación obtiene su nombre no solo porque nos dice cuán bien una arquitectura particular satisface las metas de calidad, sino que también provee ideas de cómo esas metas de calidad interactúan entre ellas, como realizan concesiones mutuas (tradeoff) entre ellas.

- ARID(An Evaluation Method for Partial Architectures)

Método conveniente para realizar la evaluación de diseños parciales en las etapas tempranas del desarrollo.

1.9. Conclusiones

A partir de los objetivos propuestos para este capítulo se definidos en la introducción se puede concluir lo siguiente:

A lo largo de este capítulo se ha expuesto las principales aspectos dentro de la disciplina, se ha definido la Ingeniería de Software Basada en Componentes, y como la misma tiene una relación importante con la arquitectura de software, al igual que un estudio del arte sobre dicho tema.

Se analizó las principales definiciones y caracterizaciones de la arquitectura de software, los patrones arquitectónicos y los niveles de abstracción de la arquitectura para ayudar a entender conceptos como: estilos arquitectónicos, marcos de trabajo, familias y líneas de productos y modelos o arquitecturas de referencia.

Se analizó además los principales parámetros de calidad que debe definir la calidad de la arquitectura de software propuesta.

Se integró la metodología de desarrollo con el objeto de estudio y la relación que existe entre ellas, así como una descripción detallada de las actividades a realizar por el arquitecto que conllevan a la realización de todas las tareas definidas en esta investigación.

CAPITULO 2

Diseño arquitectónico

2.1. Introducción

En este capítulo se hace una propuesta de solución al problema panteado en el diseño teórico de la investigación.

La solución propuesta está dividida en dos subtópicos principales que son los artefactos fundamentales que define y construye el arquitecto de software:

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

A través de estos dos artefactos se propone la solución arquitectónica de todo el sistema y en el se incluyen los demás artefactos que define la metodología y antes se expusieron en el capítulo 1.

2.2. Línea Base de la Arquitectura

2.2.1. Introducción

La Línea de Base de la Arquitectura contiene los elementos más importantes para lograr la máxima abstracción en el diseño arquitectónico de la aplicación.

Está basada principalmente en la metodología de desarrollo Proceso Unificado de Desarrollo (Rational Unified Process), aunque no es un artefacto propuesto por la metodología.

En la misma se exponen los estilos arquitectónicos para la aplicación, se expondrán los principales componentes o elementos de la arquitectura, los conectores y sus configuraciones. Se describen los principales patrones de arquitectura utilizados, las restricciones de hardware o software de la arquitectura y se describe las tecnologías y herramientas que se utilizarán en el desarrollo del sistema.

Propósito

El propósito de la Línea Base de la Arquitectura es proporcionar la información necesaria para estructurar el sistema desde el más alto nivel de abstracción. En ella se describe la estructura del sistema en cuanto a los elementos, los conectores, las configuraciones y sus restricciones.

Los usuarios de la Línea Base de la Arquitectura son:

- El equipo de arquitectos del proyecto, que le sirve de guía para la toma de decisiones arquitectónicas y son los encargados del mantenimiento y refinamiento de esta línea base.
- Los miembros del equipo de desarrollo encargados de desarrollar la aplicación, la utilizan como la guía para la implementación del sistema.
- Los clientes tienen en ella una garantía de la calidad y el conocimiento sobre en que tecnología está desarrollada su solución.

Alcance

La Línea Base de la Arquitectura describe la estructura de la aplicación en su más alto nivel de abstracción.

Describe detalladamente el organigrama de la arquitectura encarnada en los estilos arquitectónicos que se utilizarán; los principales frameworks de desarrollo y como se adaptarán a la solución; se propone la utilización de un conjunto de patrones que resuelven problemas que se podrían presentar a lo largo del desarrollo del sistema.

Se proponen las principales tecnologías y herramientas que soportan los estilos y patrones especificados y cumplen con las restricciones del sistema.

2.2.2. Concepciones generales

El sistema se desarrollará a través de la metodología de desarrollo de software Proceso Unificado de Desarrollo (Rational Unified Process) lo que define tres elementos fundamentales que conforman la guía para el desarrollo de la arquitectura como parte de la solución.

- Guiado por los Casos de Uso
- Centrado en la arquitectura
- Iterativo e incremental

Que el desarrollo de la solución sea guiado por lo casos de uso permitirá la configuración del sistema en cuanto a módulos y subsistema principalmente teniendo en cuenta la prioridad y la complejidad de los distintos casos de usos y la seguridad de que el producto final satisfaga los requerimientos reales del cliente.

Cada una de las fases en el desarrollo definidas por la metodología de desarrollo termina con la consecución de un conjunto de hitos, la terminación de cada uno de ellos no se pretende que se realicen de una vez sino que se vayan alcanzando a medida que se vayan refinando, iterando una y otra vez de acuerdo con el plan de iteraciones definidos en el Plan de Desarrollo del Sistema.

2.2.3. Organigrama de la arquitectura

Como se había visto en el capítulo anterior la arquitectura de software es la estructura del sistema desde distintos niveles de abstracción, encarnado en sus elementos, conectores, sus configuraciones y restricciones.

Los estilos arquitectónicos son el nivel de abstracción mayor para estructurar el sistema, la elección del mismo está dado por el tipo de aplicación que se vaya a desarrollar.
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, la que se transforma para facilitar las operaciones comerciales y posibilitar la toma de decisiones en cuanto a las principales operaciones[Pressman, 2000]:

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 debe realizar las operaciones convencionales de procesamiento de datos, los que crea una lógica de negocio muy densa.

Los sistemas de inventarios son sistemas que su lógica organizacional pueden cambiar con gran rapidez por tanto entre más modular y distribuido este su código será mucho mejor para su posterior mantenimiento.

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

Estilo Arquitectura en Capas

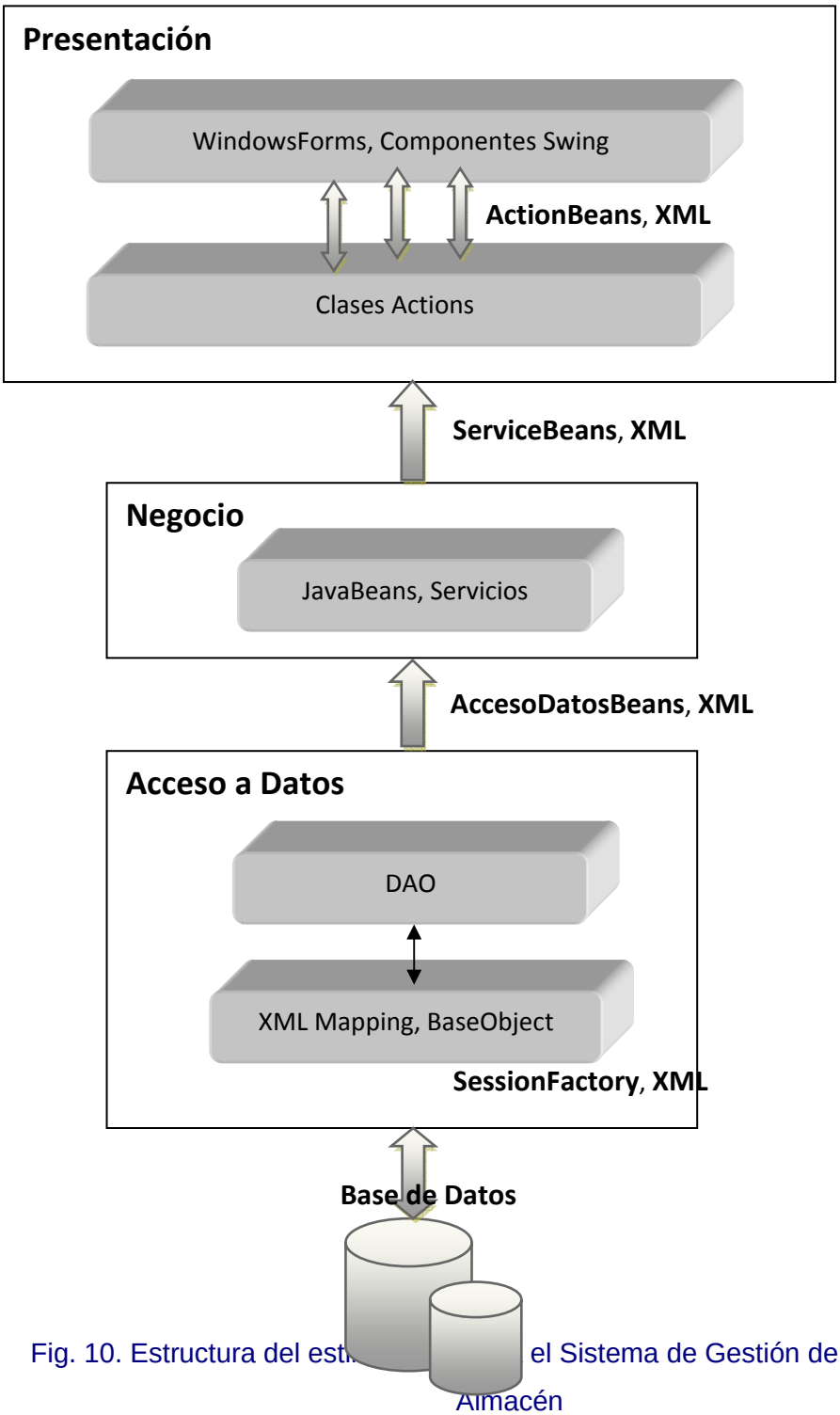


Fig. 10. Estructura del esti el Sistema de Gestión de Inventario y Almacén

Componentes o elementos

Los principales componentes (elementos) que distinguen este estilo arquitectónico son las capas:

- Presentación

Esta capa se encuentra dividida en dos subcapas fundamentales que entre las dos contienen la funcionalidad necesaria para permitir que los clientes interactúen e intercambien información con la aplicación:

1. Interfaz de Usuario

   La subcapa Interfaz de Usuario tiene la funcionalidad necesaria para que los usuarios intercambien información con la aplicación. Sus principales componentes son las WindowsForm y los componentes desarrollados a partir del framework Swing.

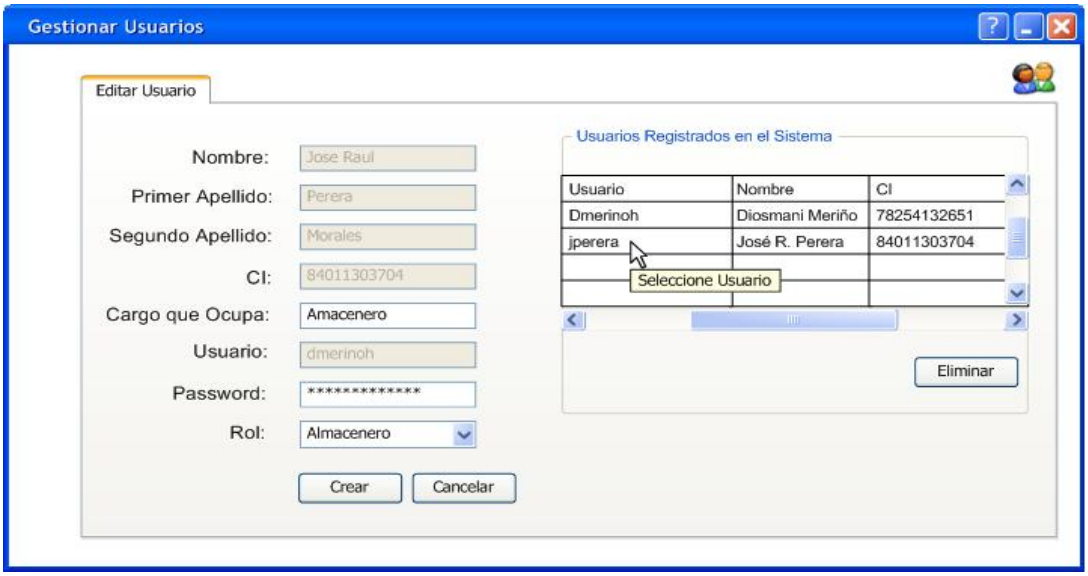


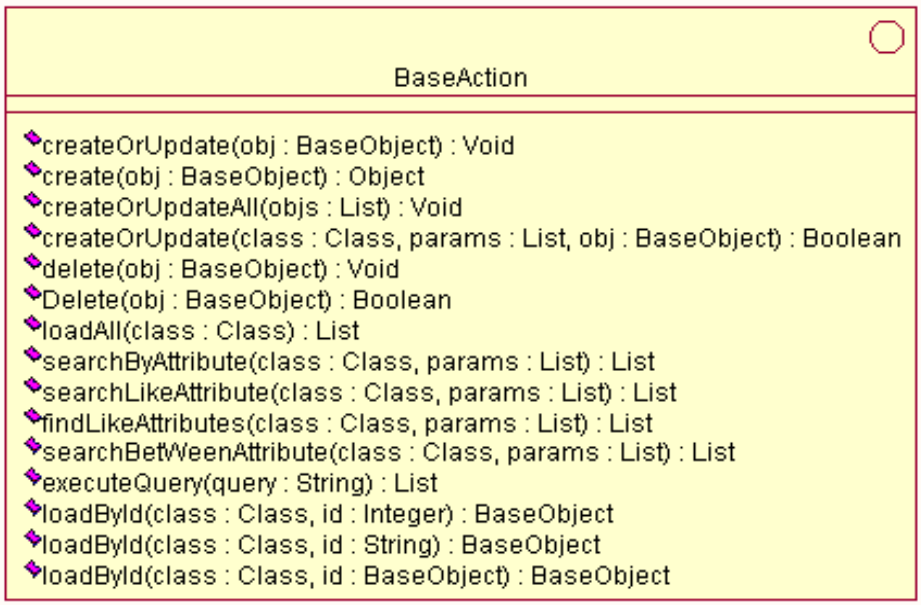
Fig. 11. Interfaz de usuario

2. Actions (Acciones)

   Tiene la funcionalidad de encapsular la lógica para la presentación de la información al usuario, es la encargada de gestionar los eventos y la principal relación con el “modelo” de los datos que se deben mostrar. Aquí se hacen las principales validaciones de los datos a entrar en el sistema.

   Sus principales componentes son los ActionsBeans.

Ejemplo:



Cualquier implementación de esta interface contendría una implementación de cada uno de estos métodos, en el caso que exista la necesidad de incorporar nuevas funcionalidades que con estas no se resuelvan se puede crear un action específico para esa situación.

3. Conectores entre las subcapas Action e Interfaz de Usuario

A partir de las clases Actions (Acciones) se crea un ActionBean utilizando el lenguaje XML (Lenguaje de Marcado), creando los JavaBeans para la presentación, se logra integrar las interfaces de usuario con las clases que controla las acciones.

Ejemplo:

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

Se crea el Bean correspondiente a la WindowsForm de autenticar enAutenticarForm y se le referencia por propiedad el Bean de la clase Action autenticarAction que va a controlar las acciones de esta WindowsForm.

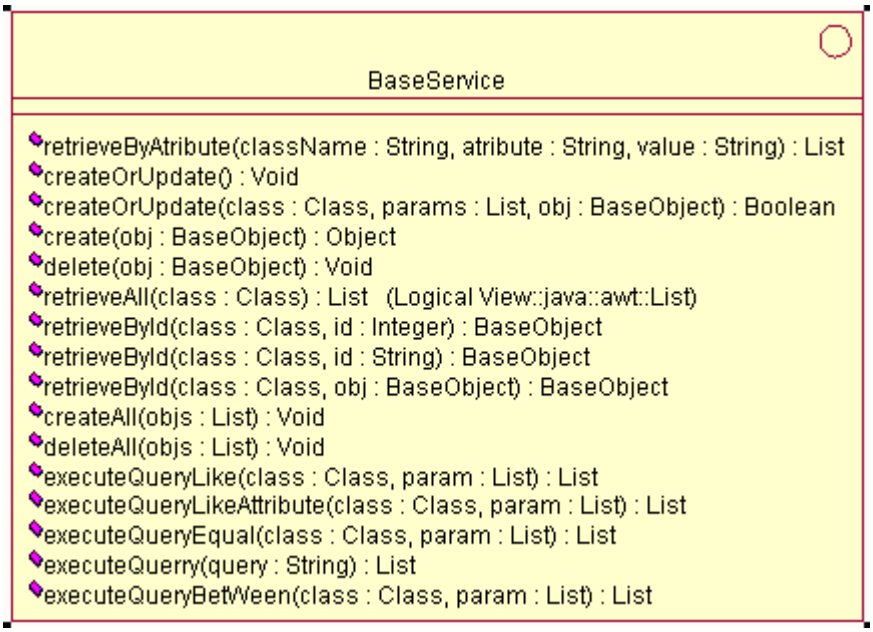
➢ Negocio

Esta capa almacena la lógica del negocio, en la implementación de cada una de las clases que la integran se manejan todas las transacciones de la aplicación y se aplican cada una de las reglas del negocio.

1. Servicios

   Los Servicios son los componentes principales de esta capa, son las clases que encapsulan las funcionalidades del negocio de la aplicación y que ejecutan las transacciones, cálculos y devuelven la información que se necesita.

   Ejemplo:



   Cualquier implementación de esta Interface contendría una implementación de cada uno de estos métodos, en el caso que exista la necesidad de incorporar nuevas funcionalidades que con estas no se resuelvan se puede crear un servicio específico para esa situación.

➢ Acceso a Datos

Esta capa contiene la funcionalidad de acceder al repositorio de los datos, es la encargada de ejecutar la lógica de acceso a los datos; tiene dos subcapas fundamentales:

1. XML Mapping (Ficheros de Mapeo)

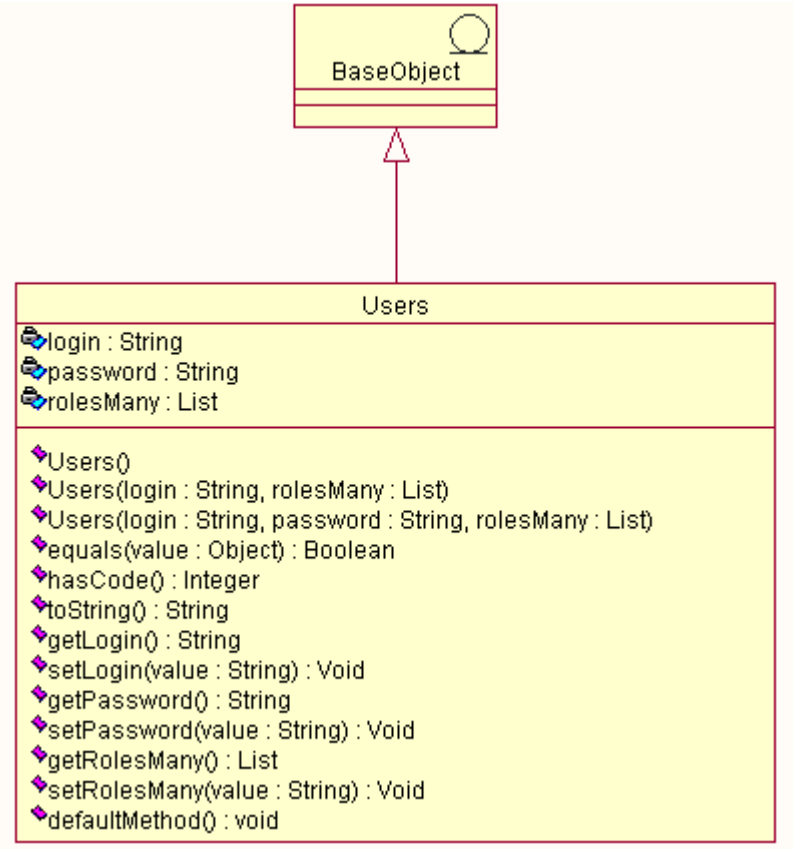
Los ficheros de mapeo, son ficheros XML que contienen la información de la Base de Datos a la que hace referencia, contiene los campos de cada una de las tablas y sus relaciones.

Ejemplo:

```
<hibernate-mapping schema="dbo" package="zun.inventario.commons.model.data">
  <class name="Users" table="users" schema="dbo" optimistic-lock="none" lazy="false">
    <id name="login" type="string" unsaved-value="null">
      <column name="login" not-null="true" unique="true" index="PK1" length="50"/>
      <generator class="assigned"/>
    </id>
    <property name="password" type="string" column="password" length="32"/>
    <set name="rolesMany" table="users_rol" cascade="none" lazy="false">
      <key foreign-key="Refusers65">
        <column name="login" not-null="true" index="PK8" length="50"/>
      </key>
      <many-to-many class="zun.inventario.commons.model.data.Roles" foreign-key="Refrole
        <column name="id_rol" not-null="true" index="PK8" length="50"/>
      </many-to-many>
    </set>
  </class>
</hibernate-mapping>
```

2. Objetos de Negocio, BaseObject

Los Objetos de Negocio que se generan a partir del mapeo de las clases de las tablas de la base de datos son clases entidades de Java que contiene los atributos y métodos para acceder a ellos, estos constituyen los objetos con que se trabaja en la aplicación.

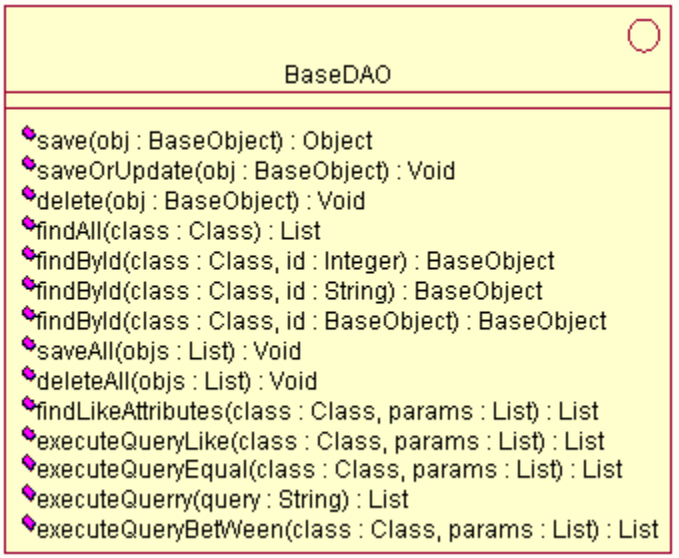


Todas los objetos del negocio heredan de la clase BaseObject, con el fin de homogeneizar todos los objetos.

3. DAO (Objetos de Acceso a Datos)

Los Objetos de Acceso a Datos son las clases que contienen la funcionalidad necesaria para acceder a la Base de Datos y realizar un conjunto de operaciones definidas para poder realizar las transacciones.

Ejemplo:



Cualquier implementación de esta Interface contendría una implementación de cada uno de estos métodos, en el caso que exista la necesidad de incorporar nuevas funcionalidades que con estas no se resuelvan se puede crear un DAO específico para esa situación.

Conectores / Configuraciones

Los conectores son las formas de comunicación entre los distintos componentes o elementos definidos en el estilo arquitectónico, los conectores que se utilizaron para unir las distintas capas son:

- Presentación – Negocio, ServiceBeans

Los Servicios que se crean en la capa de Negocio son referenciados por los ActionBeans creados en la capa de Presentación.

La referencia a estos Beans puede hacerse mediante el Constructor o mediante una propiedad. De esta forma los ActionBeans pueden acceder a las funcionalidades que les brindan los servicios referenciados.

Ejemplo:

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

En este caso se le referencia al bean zunBaseAction el bean del servicio zunService por el constructor, lo que quiere decir que al invocar al constructor de la clase Action se está invocando al bean del servicio. De esta forma quedan conectadas las dos capas.

El esquema utilizado para los XML donde se referencian los Beans es el propuesto por el framework Spring:

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

- Negocio – Acceso a Datos, ServiceBeans

Los DAO que se crean en la capa de Acceso a Datos son referenciados por los bean que se crean en la capa de Negocio (clases Servicios).

La referencia a estos Beans puede hacerse mediante el constructor o mediante una propiedad. De esta forma los ServiceBeans pueden acceder a las funcionalidades que les brindan los DAO referenciados.

Ejemplo:

```
<bean id="zunBaseService"
    class="zun.inventario.core.service.impl.ZunBaseServiceImpl">
    <constructor-arg>
        <ref local="zunBaseDAO" />
    </constructor-arg>
</bean>
```

En este caso se le referencia al Bean zunBaseService el Bean del DAO zunBaseDAO por el constructor, lo que quiere decir que al invocar al constructor de la clase del Servicio se está invocando al bean del DAO. De esta forma quedan conectadas las dos capas.

- Acceso a Datos – Base Datos, SessionFactory

El SessionFactory es un Bean del Framework Hibernate que proporciona 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>
```

El SessionFactory es la fábrica de las sesiones (conexiones) que se crean a la base de datos, contiene los datos necesarios para realizar la conexión los que se le pasan como propiedad al bean, por ejemplo: el driver que identifica a que tipo de base de datos se va a conectar, el usuario y el password para conectarse y el nombre del servidor.

El schema del XML es el definido por el framework:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//E
 "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
```

Restricciones

La principal restricción que se le impone a los componentes de este estilo arquitectónico es:

- Los componentes que pertenecen a las capas superiores solo pueden hacer uso de los componentes de sus capas inmediatas inferiores.

La organización de los componentes en cada una de las capas es la principal función del estilo de arquitectura en Capas.

2.2.4. Frameworks de desarrollo

Otro de los niveles de abstracción que se ha visto en el capítulo anterior fueron los Frameworks, que no son más que arquitecturas definidas para un determinado dominio de la aplicación que contiene un conjunto de componentes implementados y sus interfaces bien definidas, estos componentes se pueden utilizar, redefinir y crear nuevos componentes.

A partir de la organización estructural del sistema en Capas se propone la utilización de tres frameworks fundamentales, distribuidos en cada capa de la aplicación con funcionalidades bien definidas y componentes implementados que permiten desarrollar nuestros propios componentes para cada capa de la aplicación:

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

Framework Swing

Características [Gomez, 2006]

- Amplia variedad de componentes: En general las clases que comiencen por "J" son componentes que se pueden añadir a la aplicación. Por ejemplo: JButton.

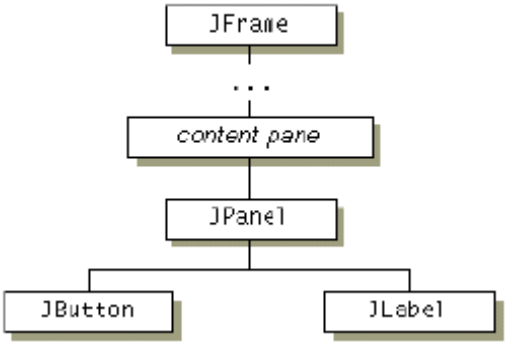


Fig. 12. Ejemplo de Jerarquía de clases de Swing

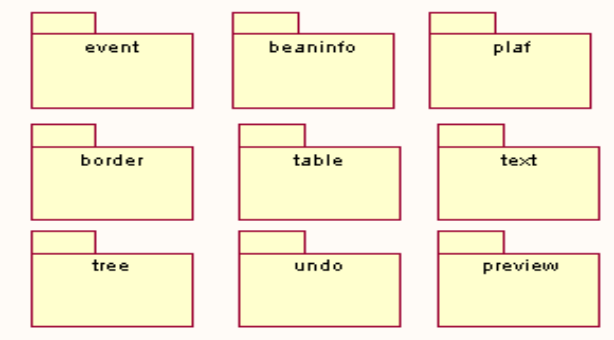
- Aspecto modificable (look and feel): Se puede personalizar el aspecto de las interfaces o utilizar varios aspectos que existen por defecto (Metal Max, Basic Motif, Window Win32).
- Arquitectura **Modelo-Vista-Controlador**: Esta arquitectura da lugar a todo un enfoque de desarrollo muy arraigado en 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. Son hijos de ActionListener.

- 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.
- Bordes complejos: Los componentes pueden presentar nuevos tipos de bordes. Además el usuario puede crear tipos de bordes personalizados.
- Diálogos personalizados: Se pueden crear multitud de formas de mensajes y opciones de diálogo con el usuario, mediante la clase JOptionPane.
- Clases para diálogos habituales: Se puede utilizar JFileChooser para elegir un fichero, y JColorChooser para elegir un color.
- 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.
- Capacidad para "deshacer": En gran variedad de situaciones se pueden deshacer las modificaciones que se realizaron.
- 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.

Modelo del Framework

El Framework Swing tiene el siguiente modelo de paquetes donde contiene los modelos diseño de componentes del Framework [Rational, 2003]



- Paquete **beanInfo**

Contiene clases que manejan los Bean

- Paquete **event**

Contiene un conjunto de interfaces que permiten el control de los eventos sobre los componentes visuales.

- Paquete **plaf**

Contiene los paquetes que definen las clases con los estilos en que se muestran los componentes visuales.

- Paquete **border**

Contiene las clases que manejan los estilos de Bordes de los componentes.

- Paquete **table**

Contiene el conjunto de clases que permiten crear tablas, estilos de tablas y sus formas.

- Paquete **text**

Estos dos paquetes encierran las clases necesarias para trabajar los documentos en formato RTF o html.

- Paquete **tree**

Conjunto de clases e interfaces que permiten el tratamiento del componente árbol

- Paquete **undo**

Conjunto de clases e interfaces que permiten la funcionalidad de revertir las operaciones.

- Paquete **preview**

Contiene un paquete que contiene las clases necesarias para el tratamiento de ficheros en los distintos tipos de Sistemas Operativos.

Ejemplo de algunas de las clases más importantes:

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

- ***ButtonGroup***: Muestra una lista de elementos (JRadioButton) con solo uno seleccionable. Cada elemento tiene un círculo, que en caso del elemento seleccionado contendrá un "punto".
- ***JToggleButton***: Es como un botón normal, pero al ser pinchado por el usuario queda activado.
- ***JProgressBar***: Representa una barra de estado de progreso, mediante la que habitualmente se muestra el desarrollo de un proceso en desarrollo (ejemplo: la instalación de una aplicación).
- ***JTabbedPane***: Es una ventana con solapas (la que utiliza Windows). Este componente había sido muy solicitado.
- ***JApplet***: Aunque ya existía una clase Applet en AWT, esta nueva versión es necesaria para crear applets Java que utilicen interfaces Swing.

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, te facilita el desarrollo de nuevas funcionalidades y hace que la curva de aprendizaje sea favorable para el desarrollador, al ser fácil de asimilar. [SPR, 2007]

El objetivo central de Spring es permitir que objetos de negocio y de acceso a datos sean reusables, no atados a servicios J2EE específicos. Estos objetos pueden ser reutilizados tanto en entornos J2EE (web o EJB), aplicaciones standalone y entornos de pruebas.

Filosofía de Spring Framework

- Proveer un Framework no invasivo.
- Siempre que se pueda reusar código.
- Plug & Play de componentes.

Dentro de las ventajas que ofrece Spring, se encuentran que facilita la manipulación de nuestros objetos, se usen EJBs o no, reduce la proliferación de Singletons (Patrón de Diseño Singleton), elimina la necesidad de usar distintos y variados tipos de ficheros de configuración, mejora la práctica de programación, permite el uso o no de EJBs, realizando el mismo tipo de funciones sin ellos.

Spring proporciona:

- Una potente 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
<?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:

```xml
<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 de 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.

Framework Hibernate 3.2 [Bauer, 2005]

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
Session	Es la más usada en las aplicaciones, es la manejadora de la persistencia, a través de ella podremos guardar y cargar objetos de la base de datos.
SessionFactory	Mediante ella se obtiene la Session.
Configuration	Es usada para configurar Hibernate se utiliza para especificar la ruta de los documentos de mapeo.
Transaction	Se utiliza para el control de las transacciones.
Query y Criteria	Permiten la ejecución de consultas a la Base de Datos.

Tabla 2. Interfaces del framework Hibernate

Además de estas interfaces Hibernate brinda otros aspectos muy importantes y que serán de gran utilidad en nuestras aplicaciones, por ejemplo:

- La interface *Type*, es un elemento fundamental y muy poderoso, permite el mapeo de tipos "java" a columnas en bases de datos incluso a varias columnas, incluye tipos como *Calendar*, *byte[],* y permite además definir nuestros propios tipo de datos (Persona, Dirección, Nombre).
- Las interfaces *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, puesto que es usado solamente para extraer datos no para borrar, insertar o actualizar, nos permite realizar complejas consultas.

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

Define dos tipos de configuración:

- Entorno manejado, 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.

En los entornos no manejados Hibernate se ocupa de las transacciones y las conexiones JDBC o lo deja para que el desarrollador se ocupe de esto, en el caso de los entornos manejados Hibernate se integra con el contenedor para ocuparse de los DataSources y de las transacciones. Salvo estas diferencias lo demás es común para cualquier entorno.

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 pero además se trata todo lo referente a las relaciones de esta clase con otras incluyendo el tratamiento de herencia y polimorfismo. Ahora el archivo de configuración es el que le dice a Hibernate todo 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
<?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.

```java
import org.apache.commons.lang.builder.EqualsBuilder;
import org.apache.commons.lang.builder.HashCodeBuilder;
import org.apache.commons.lang.builder.ToStringBuilder;
import zun.inventario.core.model.BaseObject;

public class AUsuario extends BaseObject{

    private String usuario;
    private ARol rol;
    private String contrasena;
    private String nombreU;
    private String primerApellido;
    private String segundoApellido;
    private String carnetIdentidad;
    private String cargoUsuario;

    public AUsuario() {
        super();
    }
    public AUsuario(String usuario) {
        super();
        this.usuario = usuario;
    }
    public AUsuario(String usuario, String contrasena, String nombreU,
            String primerApellido, String segundoApellido,
            String carnetIdentidad, String cargoUsuario) {
        super();
```

La clase de Java *AUsuario* correspondiente al fichero de mapeo A_Usuario.

Ventajas:

- Abstrae del manejo del código SQL, permitiendo ganar en tiempo de desarrollo.
- No necesita ningún entorno especial para correr sea servidor aplicaciones, contenedores, etc.
- Las clases que persisten no tienen que implementar ninguna interfaz especial ni extender ninguna clase, por lo que son completamente trasparentes haciendo posible su reutilización en distintas partes de nuestra aplicación con el objetivo de transportar los datos, y haciéndola desacoplada.
- Permite manejar las transacciones de una manera eficaz.
- Toda la configuración de la conexión y el mapeo se realiza en ficheros externos, haciendo transparente el uso de distintos gestores
- Permite manejo de almacén de conexiones (Connections Pool).
- Es un framework open source y gratis.

Desventajas

- No es un estándar J2EE.
- Es un framework complejo. Puede costar un poco de esfuerzo dominar cada una de las propiedades que caracterizan al mismo.

2.2.5. Patrones de Arquitectura

Se entiende por patrón una solución probada que se puede aplicar con éxito a un determinado tipo de problemas que aparecen repetidamente en algún campo. El establecimiento de estos patrones comunes es lo que posibilita el aprovechamiento de la experiencia acumulada en el diseño de aplicaciones.

Los siguientes son los patrones propuestos, divididos por capas en las principales categorías de patrones arquitectónicos:

- Capa de Presentación
  1. Modelo Vista Controlador

     Nombre: Modelo - Vista – Controlador

Problema: La lógica de un interfaz de usuario cambia con más frecuencia que los almacenes de datos y la lógica de negocio. Si realizamos un diseño complicado, es decir, una mezcla de los componentes de interfaz y de negocio, entonces la consecuencia será que, cuando se necesite cambiar el interfaz, se tendrá que modificar trabajosamente los componentes de negocio. Mayor trabajo y más riesgo de error. [Welicki, 2007]

Solución: Se trata de realizar un diseño que desacople la vista del modelo, con la finalidad de mejorar la reusabilidad. De esta forma las modificaciones en las vistas impactan en menor medida en la lógica de negocio o de datos.

Elementos del patrón:

- Modelo: datos y reglas de negocio.
- Vista: muestra la información del modelo al usuario.
- Controlador: gestiona las entradas del usuario.

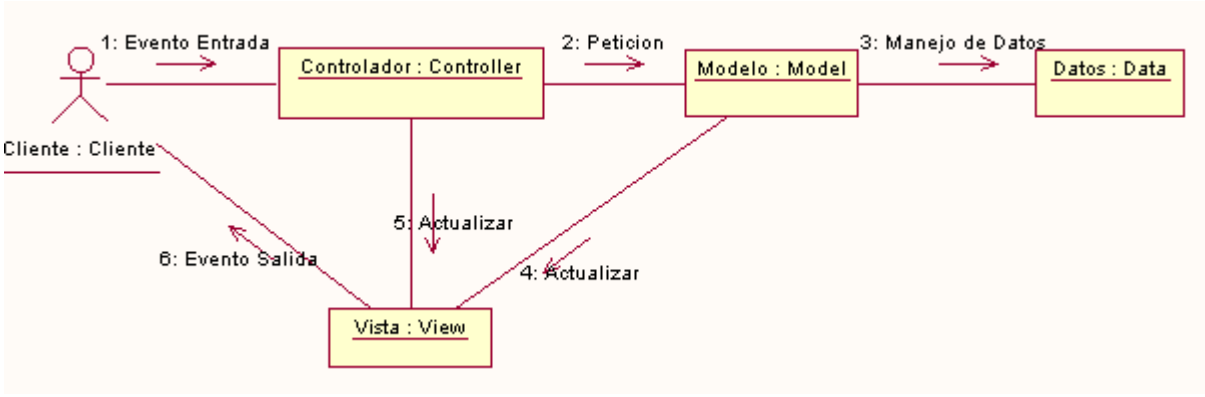


Fig. 13. Estructura del patrón Modelo – Vista- Controlador [Palos, 2006]

En el sistema se deberá utilizar un modelo pasivo (aquel que no notifica cambios en los datos) es decir que responde a las entradas de los usuarios, pero no detecta los cambios en datos del servidor, esto se hace con el objetivo de que la aplicación no caiga en un continuo cambio de las interfaces.

El framework Swing garantiza en todas sus clases la utilización de dicho patrón.

- Capa de Negocio

1. J2EE Patterns [Deepak Alur, 2001]

Nombre: Service Locator

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

Solución: Se hace necesario acceder a los servicios en distintos lugares de la aplicación, para evitar que se creen servicios ya creados y que se estandarice la forma de acceder a ellos es que se utiliza dicho patrón.

Elementos del patrón:

- ServiceCliente: Cualquier parte de la aplicación que necesite acceder a un servicio determinado.
- ServiceLocator: Componente que aplica el patrón de diseño Singleton para ser creado solo una vez en el contexto de la aplicación, tiene la funcionalidad de localizar mediante la ServiceFactory que busca o crea el servicio indicado.
- Servicio: Componente de la Capa de Negocio que contiene la lógica de negocio.

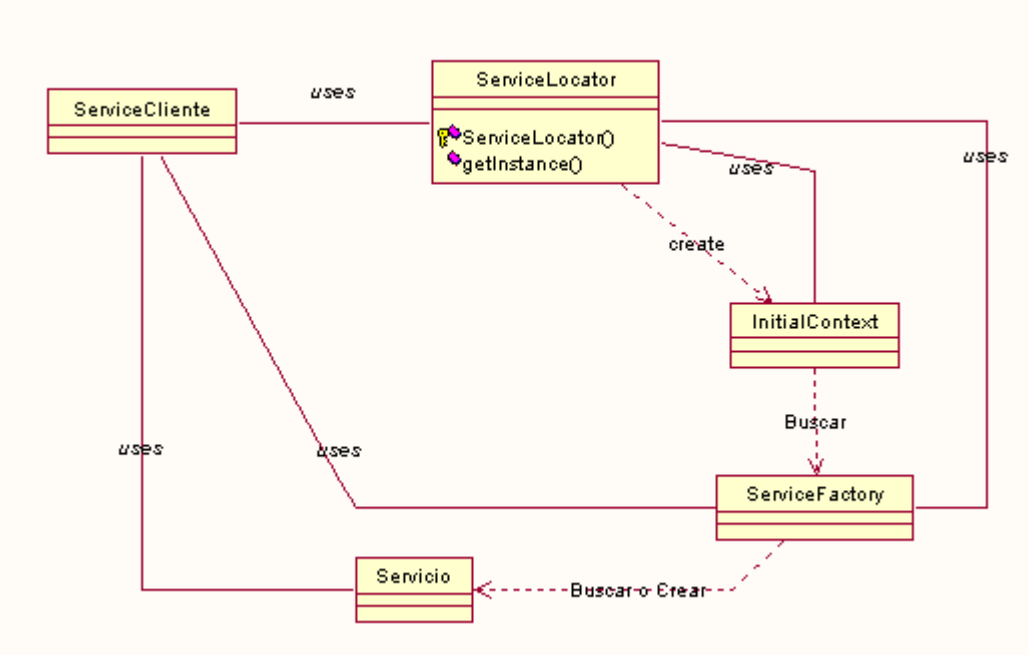


Fig. 14. Estructura del patrón Service Locator [Palos, 2006]

Implementación:

```
public class ServiceLocator extends BaseAppContext{

    public ServiceLocator(){
        super();
    }

    public Object lookupService(String serviceName){
        return getAppContext().getBean(serviceName);
    }

}
```

- Capa de Acceso a Datos.

1. J2EE Patterns [Deepak Alur, 2001]

Nombre: Data Access Object (DAO)

Problema: El acceso a los datos varía dependiendo de la fuente de los datos. Tener acceso al almacenaje persistente, por ejemplo a una base de datos, varía grandemente dependiendo del tipo de almacenaje (bases de datos emparentadas, bases de datos orientadas al objeto, ficheros " planos ", y así sucesivamente).

Solución: La creación de los Objetos de Acceso a Datos (DAO) permite acceder a los datos a través de estos objetos y no tener que cambiar los objetos de negocio.

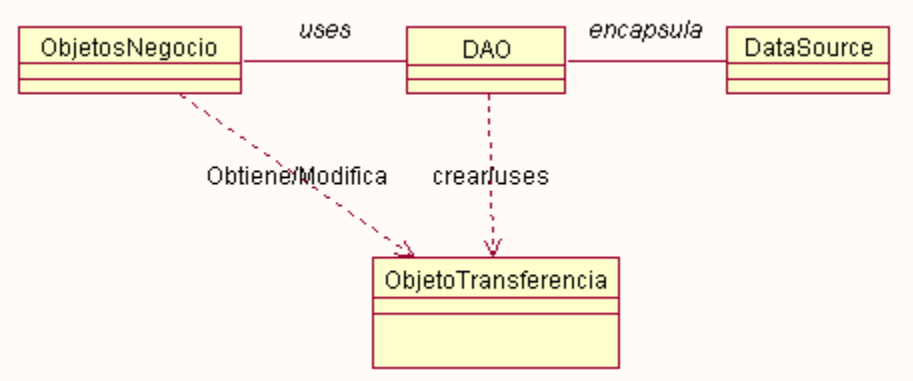
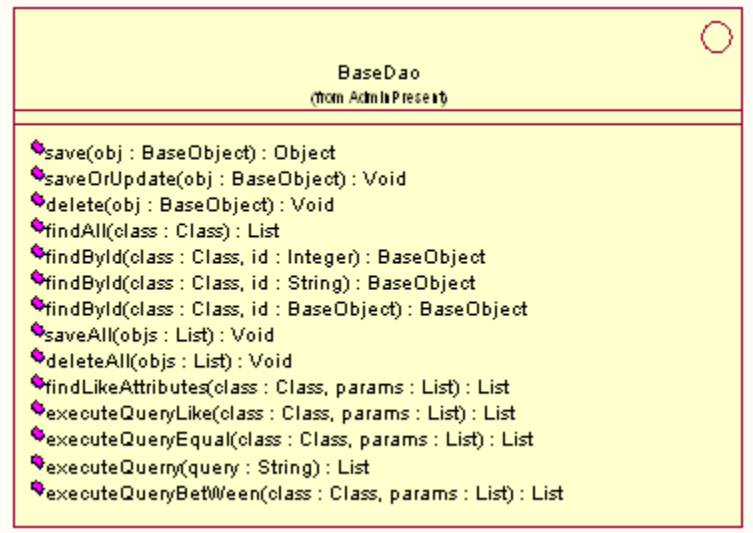


Fig. 14. Estructura del patrón Data Access Object [Palos, 2006]

Elementos del patrón:

- DAO: Los DAO encapsulan la funcionalidad para: insertar, update, eliminar y otras por ejemplo:



BaseDao contiene la funcionalidad necesaria para acceder a los datos del repositorio de datos.

2.2.6. Lenguaje, tecnología y herramientas de soporte al desarrollo

Lenguaje de desarrollo

No es arriesgado afirmar que Java supone un significativo avance en el mundo de los entornos software, y esto viene avalado por dos elementos claves que diferencian a este lenguaje desde un punto de vista tecnológico:

- Es un lenguaje de programación que ofrece la potencia del diseño orientado a objetos con una sintaxis fácilmente accesible y un entorno robusto y agradable.
- Proporciona un conjunto de clases potente y flexible.

Caracteristicas de Java

- Potente
  - Orientación a Objetos

    En este aspecto Java fue diseñado partiendo de cero, no siendo derivado de otro lenguaje anterior y no tiene compatibilidad con ninguno de ellos.

    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. [Autores Colectivo, 1999]
  - 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. [Autores Colectivo, 1999]
  - 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. Se realiza un descubrimiento de la mayor parte de los errores durante el tiempo de compilación, 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, Java libera al programador del compromiso de tener que controlar especialmente la asignación que de ésta hace a sus necesidades específicas, esto a diferencia de C++. Este lenguaje posee una gestión avanzada de memoria llamada gestión de basura (Garbage Collector), y un manejo de excepciones orientado a objetos integrados. Estos elementos realizarán muchas tareas antes tediosas a la vez que obligadas para el programador como la destrucción de objetos. [Autores Colectivo, 1999]
  - Modelo de objeto rico

    Existen varias clases que contienen las abstracciones básicas para facilitar a los programas una gran capacidad de representación. Para ello se contará con un conjunto de clases comunes que pueden crecer para admitir todas las necesidades del programador.

    Además la biblioteca de clases de Java proporciona un conjunto único de protocolos de Internet.

- Simple
  - Fácil aprendizaje

    El único requerimiento para aprender Java es tener una comprensión de los conceptos básicos de la programación orientada a objetos. Así se ha creado un lenguaje simple (aunque eficaz y expresivo) pudiendo mostrarse cualquier planteamiento por parte del programador sin que las interioridades del sistema subyacente sean desveladas.

    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. [Autores Colectivo, 1999]

  - Completado con utilidades

    El paquete de utilidades de Java viene con un conjunto completo de estructuras de datos complejas y sus métodos asociados, que serán de inestimable ayuda para implementar applets y otras aplicaciones más complejas. Se dispone también de estructuras de datos habituales, como pilas y tablas hash, como clases ya implementadas.

    Existirá una interfaz Observer/Observable que permitirá la implementación simple de objetos dinámicos cuyo estado se visualiza en pantalla.

    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.

- Interactivo
  - Interactivo y animado

    Uno de los requisitos de Java desde sus inicios fue la posibilidad de crear programas en red interactivos, por lo que es capaz de hacer varias cosas a la vez sin perder rastro de lo que debería suceder y cuándo. Para se da soporte a la utilización de múltiples hilos de programación (multithread).

Las aplicaciones de Java permiten situar figuras animadas en las páginas Web, y éstas pueden concebirse con logotipos animados o con texto que se desplace por la pantalla. También pueden tratarse gráficos generados por algún proceso. Estas animaciones pueden ser interactivas, permitiendo al usuario un control sobre su apariencia. [Autores Colectivo, 1999]

- 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. [Autores Colectivo, 1999]

- Otras
  - Seguridad

    Existe una preocupación lógica en Internet por el tema de la seguridad: virus, caballos de Troya, y programas similares navegan de forma usual por la red, constituyendo una amenaza palpable. Java ha sido diseñado poniendo un énfasis especial en el tema de la seguridad, y se ha conseguido lograr cierta inmunidad en el aspecto de que un programa realizado en Java no puede realizar llamadas a funciones globales ni acceder a recursos arbitrarios del sistema, por lo que el control sobre los programas ejecutables no es equiparable a otros lenguajes.

    Los niveles de seguridad que presenta son:

    - 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.

➢ Diferentes tipos de aplicaciones de interés

En Java podemos crear los siguientes tipos de aplicaciones:

- 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.

IDE (Entorno de Desarrollo Integrado)

Eclipse

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.[Eclipse, 2007]

Eclipse fue desarrollado originalmente por IBM como el sucesor de su familia de herramientas para VisualAge. Eclipse es ahora desarrollado por la Fundación Eclipse, una organización independiente sin ánimo de lucro que fomenta una comunidad de código abierto y un conjunto de productos complementarios, capacidades y servicios.

Arquitectura

La base para Eclipse es la Plataforma de cliente enriquecido RCP(Rich Client Platform). Los siguientes componentes constituyen la plataforma de cliente enriquecido:

- 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 entorno integrado de desarrollo (IDE) de Eclipse emplea módulos (plug-in) para proporcionar toda su funcionalidad al frente de la plataforma de cliente rico, 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. Adicionalmente a permitirle a Eclipse extenderse usando otros lenguajes de programación como son C/C++ y Phyton, permite a Eclipse trabajar con lenguajes para procesado de texto como LaTeX, aplicaciones en red como Telnet y Sistema de gestión de base de datos. La arquitectura plugin permite escribir cualquier extensión deseada en el ambiente, como seria Gestión de la configuración. Se provee soporte para Java y CVS en el SDK de Eclipse. Este no debe ser usado solamente para soportar otros lenguajes de programación.

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. El IDE también hace uso de un espacio de trabajo, en este caso un grupo de metadata en un espacio para archivos plano, permitiendo modificaciones externas a los archivos en tanto se refresque el espacio de trabajo correspondiente.

Características

La versión que se utilizará en el desarrollo del Sistema será Eclipse 3.2.2 que dispone de las siguientes características: [Eclipse Foundation, 2007]

- 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

Eclipse como Herramienta de Desarrollo

El editor de Java y la compilación incremental

- El editor de Java resalta el código Java de manera que sea más fácil leerlo y editarlo y autocompletarlo mientras se escribe, como cualquier otro editor Java.
- Una de las ventajas de Eclipse es que compila el código de forma incremental, mientras escribimos. Esto permite detectar errores de compilación en tiempo de programación. El editor de Java muestra los errores de compilación según se programa, ahorrando el tiempo que se tarda en compilar y buscar las líneas de errores, pues muestra en el editor los fallos cometidos. 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.
- Otra ventaja del editor incluida en las últimas versiones es que es capaz de “esconder” el contenido de métodos o clases, mostrando solo su primera línea, funcionalidad muy util en ficheros de tamaño considerable.

- También incluye un sistema de búsqueda en todos los documentos de un proyecto, que nos permite buscar cadenas en fichero y reemplazarlas.

Herramientas de refactorización y de código

- Eclipse incluye otra funcionalidad importante que son las herramientas de refactorización. 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.
- Incluye otras herramientas para ayudarnos al escribir nuestro código. Es capaz de generar métodos get y set, y métodos que sobreescriben a los de clases superiores.
- Tiene una herramienta para ordenar el código, que realiza la tabulación del código, espacios y otros cambios siguiendo la especificación recomendada por Sun.
- Ordena las clases importadas, evitando el uso de "*" para importar paquetes completos.
- Genera plantillas para la documentación en formato Javadoc para clases, métodos o atributos, con plantillas configurables.
- Una de las funciones más útiles es la de externalización de cadenas, que permite extraer las cadenas de una clase a un fichero de propiedades, que luego pueda usarse para traducir la aplicación.

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 muestra los resultados de los tests.

Perspectiva Debug

- Eclipse incluye una herramienta de debug, en modo gráfico, que facilita el proceso de debug.

Plug-ins para Eclipse

Una de las características más importantes de Eclipse es su modularidad, pues se pueden añadir plug-ins según las necesidades de cada desarrollador.

En este caso se ha analizado utilizar varios plug-ins no incluidos en la distribución de Eclipse como:

- WindowsBuilder, constructor de interfaces gráficas. Eclipse no dispone de un editor de interfaces gráficas, aunque si existe un subproyecto para el desarrollo de este editor.

Para la implementación como IDE de desarrollo Eclipse 3.2.2 demanda como mínimo 768 MB de RAM y recomendado 1.0 GB de RAM al integrarlo con los plugin necesarios.

Para la implementación los plugin serán:

- ***WindowsBuilder***, plugin que garantiza el trabajo con componentes visuales para la capa de presentación.
- ***Hibernate Tool,*** aporta un plugin de Eclipse y una serie de tareas para Ant que facilitan un poco la vida del desarrollador. Partiendo de un esquema de base de datos esta herramienta nos crea por medio de ingeniería inversa los ficheros de configuración para Hibernate y EJB3, componentes Seam, DAO's, etc. También permite la ejecución interactiva de consultas a la base de datos a través de HQL y EJB QL3. Dispone de editores para la edición de ficheros hbm.xml y cfg.xml.
- ***JUnit,*** es un framework que permite realizar la ejecución de clases Java de manera controlada, para poder evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera. Es decir, en función de algún valor de entrada se evalúa el valor de retorno esperado; si la clase cumple con la especificación, entonces JUnit devolverá que el método de la clase pasó exitosamente la prueba; en caso de que el valor esperado sea diferente al que regresó el método durante la ejecución, JUnit devolverá un fallo en el método correspondiente.

  JUnit es también un medio de controlar las pruebas de regresión, necesarias cuando una parte del código ha sido modificado y se desea ver que el nuevo código cumple con los requerimientos anteriores y que no se ha alterado su

funcionalidad después de la nueva modificación. El propio framework incluye formas de ver los resultados (runners) que pueden ser en modo texto, gráfico (AWT o Swing).

- ***SVNeclipse***, ***SVN*** viene de Subversion que es un software de sistema de control de versiones diseñado específicamente para reemplazar al popular CVS, el cual posee varias deficiencias. Es software libre bajo una licencia de tipo Apache/BSD y se lo conoce también como SVN por ser ese el nombre de la herramienta de línea de comandos. Una característica importante de Subversion es que, a diferencia de CVS, los archivos versionados no tienen cada uno un número de revisión independiente. En cambio, todo el repositorio tiene un único número de versión que identifica un estado común de todos los archivos del repositorio en cierto punto del tiempo.

  Características

  - Un repositorio remoto donde se alojan las copias y revisiones comunes.
  - A la vez permite mantener una copia local donde se trabaja de forma independiente, cuando se cree conveniente, la copia se envia al repositorio remoto y el sistema.
  - Verifica que no existen conflictos entre las diferentes copias.
  - Otra característica es la de permitir la muestra de cambios entre diferentes versiones, aunque esto tiene una mejor efectividad entre los ficheros de texto escrito, también es posible utilizarlo para ficheros binarios.
  - Puede ser servido, mediante Apache, sobre WebDAV/DeltaV.
  - Maneja eficientemente archivos binarios (a diferencia de CVS que los trata internamente como si fueran de texto).
  - El creado de ramas y etiquetas es una operación más eficiente; Tiene costo de complejidad constante (O(1)) y no lineal (O(n)) como en CVS.

JDK (Java Development Kid)

La JVM es un intérprete que convierte el ByteCode compilado de Java en el código de máquina nativo en que debería ejecutarse no es totalmente falso, pero es más correcto decir que es un emulador de la ejecución de un código nativo. La tecnología de máquinas virtuales se ha

distinguido de la tecnología de interpretación básicamente por el nivel en que se realiza la interpretación. Los intérpretes trasladan un "código de bit" nativo directamente a llamadas del sistema o instrucciones de máquina, pero la emulación consiste en permitir una arquitectura de máquina intermedia.

La arquitectura de la JVM se basa en el concepto de una implementación que no es específica de una máquina. Esto es, la arquitectura misma no asume nada acerca de la máquina o de las características físicas y de construcción sobre la cual es implementada. De esta manera, la JVM es una entidad autónoma e única que ejecuta los archivos de clases. La JVM se separa en cinco unidades de funcionalidad distintas, las que se dedican a la tarea de ejecutar los archivos de clases.

Memoria

El efecto de maximizar la velocidad de ejecución y minimizar el consumo de memoria es uno de los puntos de mayor atención, sobre todo por la restricción de memoria RAM especificado por los clientes del sistema.

En el modelo estándar de Java, junto con el espacio necesario para el código del usuario existe también un cuerpo sustancial de código fijo para la JVM y el entorno de ejecución. Este código incluye, por ejemplo, bibliotecas estándar, código para carga de nuevas clases, un recolector de basura y un posible intérprete de máquina virtual.

En algunas ocasiones se puede necesitar acceder a clases dinámicamente cargadas y a grandes bibliotecas de clases. En un ambiente de escasa memoria, como es el caso, sería deseable dejar fuera algo de este código de apoyo. Por ejemplo, si no hay necesidad de cargar nuevas clases en tiempo de ejecución, el código para leer archivos de clases e interpretar bytecodes, podría omitirse.

Dadas estas diferencias, los espacios mínimos de memoria de ejecución y de código serían:

- Aplicación Mínima: 32k ROM, 8k RAM.
- Aplicación Mínima con código dinámico: 128k ROM, 32k RAM.

La JDK a utilizar en la implementación del sistema sería la JDK 1.6, liberada por la SUN Microsystem que contiene un conjunto de interfaces que permiten el manejo de componentes de interfaz visual y redefine los componentes existentes con nuevas funcionalidades.

Sistema Gestor de Base de Datos

El Sistema Gestión de Inventarios y Almacén, como Software de Gestión que es, necesita un repositorio de información, donde almacenar los datos necesarios para realizar las distintas operaciones.

La utilización del estilo de Arquitectura en Capas permite que se abstraiga la lógica del negocio al almacenamiento de los datos, la capa de acceso a datos contiene toda la lógica de acceso, ya sea consultas y procedimientos almacenados, dejando a la Base de Datos como simple almacén de datos sin ninguna lógica.

La simple restricción que se le ha impuesto a la selección del Sistema Gestor de Base de Datos (SGBD) que se utilice, sería solo, la implementación de Base 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.

2.3. Documento Descripción de la Arquitectura

2.3.1. Introducción

Propósito

Este documento proporciona comprensión arquitectónica global del sistema, usando un número de vistas arquitectónicas diferentes, se muestra diferentes aspectos del sistema. Este documento permite capturar y tomar decisiones arquitectónicamente significativas que deban ser tomadas en el desarrollo del sistema.

Los propósitos que se persigue con dicho documento son:

- Mejor comprensión del sistema.
- Organización del desarrollo.
- Fomento de la reutilización.
- Hacer evolucionar el sistema.

Debe capacitar a los desarrolladores, directivos, clientes, otros usuarios para comprender con suficiente detalle y para facilitar su propia participación.

Alcance

Este documento influye en la toma de decisiones arquitectónicas con respecto al Sistema de Gestión de Inventario y Almacén. Abarca todo el Sistema de Gestión de Inventario y Almacén.

2.3.2. Representación arquitectónica

Este documento presenta la arquitectura como una serie de vistas:

- Vista de Caso de Uso
- Vista Lógica
- Vista de implementación
- Vista de despliegue

Estas vistas están representadas en UML usando Rational Rose 2003.

2.3.3. Metas y restricciones arquitectónicas

Las metas y restricciones del sistema en cuanto a seguridad, portabilidad y reusabilidad, que son significativas para la arquitectura son:

2.3.3.1. 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.

2.3.3.2. 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

2.3.3.3. Redes

1. La red existente en las instalaciones debe de soportar la transacción de paquetes de información de al menos unas 10 máquinas a la vez.
2. Para hacer más fiable 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.

2.3.3.4. Seguridad

1. La seguridad se tratará desde las primeras fases de desarrollo del sistema.
2. Se utilizará las reglas de la “programación segura”, se deberá hacer un fuerte tratamiento de excepciones.

3. Parte de la seguridad corre por parte de la tecnología Java utilizada para el desarrollo de la aplicación.
4. Se deberá garantizar la seguridad de la Base de Datos, no se permitirá que nadie tenga privilegios de administración sobre la base de datos.
5. Programación de disparadores (Triggers) en la Base de Datos para no permitir la manipulación de los datos directamente en el SGBD.
6. Se deberá utilizar usuarios de base de datos con roles bien definidos para cada una de las operaciones del sistema.
7. La asignación de usuarios y sus opciones sobre el sistema se garantizará desde el modulo de Administración del sistema.
8. Debe quedar constancia de quién, desde donde, y cuando se realizó una operación determinada en el sistema.

2.3.3.5. 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, sobre todo en las PC clientes.
3. Debido a los cambios en las condiciones económicas del país, las empresas cubanas toman decisiones continuas que cambian las condiciones en que se desarrollan los procesos, por lo que el sistema deberá implementar la forma de adaptarse ante el cambio de dichas condiciones.
4. es para la funcionalidad del sistema.
5. Deberá implementar servicios que estén a la escucha del pedido de información de otros sistemas que en el futuro requerirán información.
6. Se desarrollará cada pieza del sistema en forma de componentes (elementos) con el fin de reutilizarlos para futuras versiones del sistema.
7. La documentación de la arquitectura deberá ser reutilizable para poder documentarla como una familia de productos.

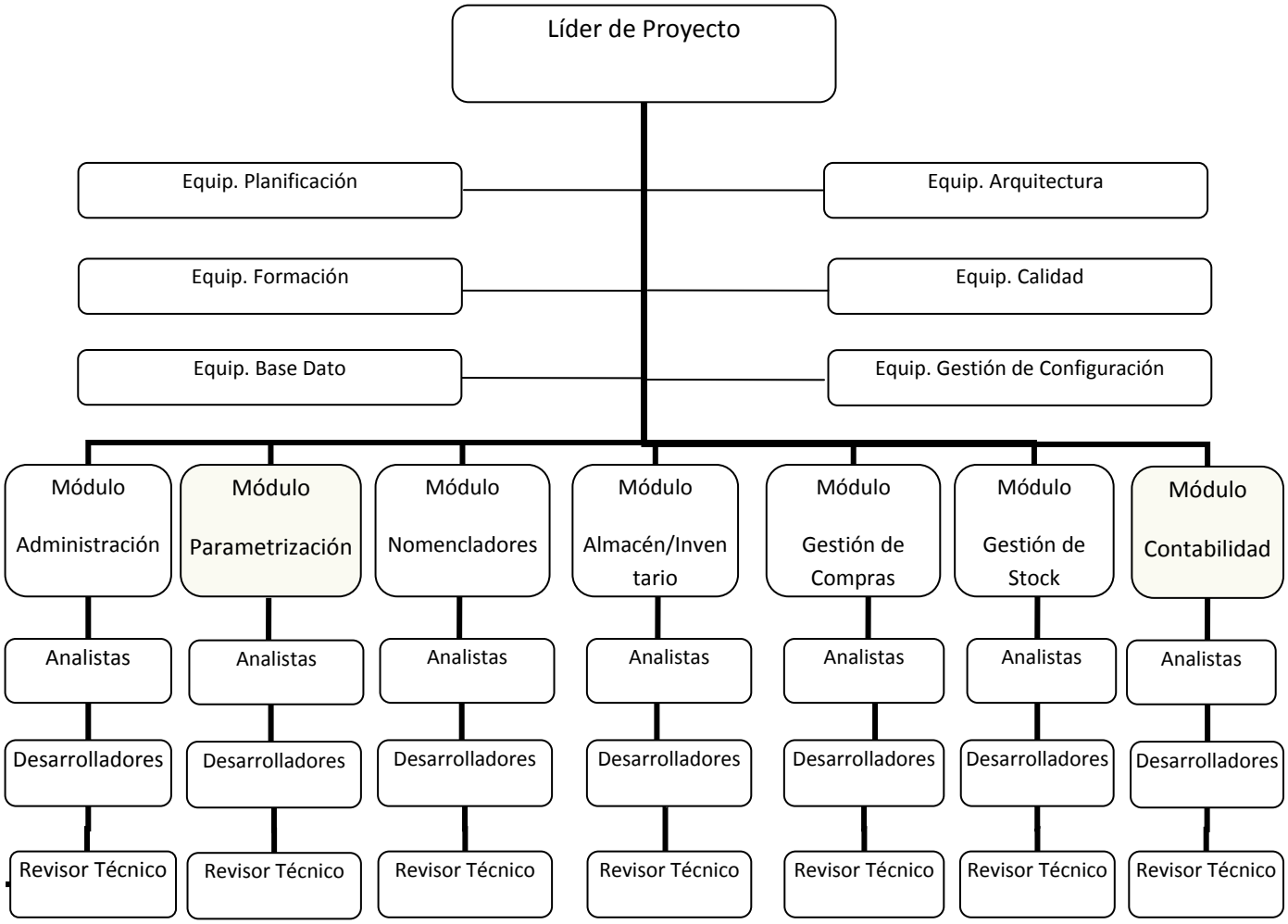
2.3.3.6. Restricciones de acuerdo a la estrategia de diseño

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

2.3.3.7. Herramientas de desarrollo

1. Para modelado Rational Rose 2003 el cual demanda como mínimo 256 MB de RAM y recomendado 512 MB de RAM.
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.
   - Capabilidades 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.

2.3.3.8. Estructura del equipo de desarrollo



El equipo de desarrollo está dividido en los principales módulo identificados para el desarrollo del sistema, compuesto por una generalización de los roles propuestos por la metodología de desarrollo Rational Unified Process (RUP): Analistas, Desarrolladores y revisor técnico.

Configuración de los puestos de trabajo por Roles:

- Rol Analista:
  - PC con mouse y teclado

- Instalación Suite de Rational 2003, para la modelación del sistema
- Paquete Office para la documentación del sistema.

- Rol Desarrollador
  - PC con mouse, teclado, 1 GB de memoria RAM
  - Instalación de la Máquina Virtual de Java (JDK 1.6)
  - Eclipse 3.2 con los plugins necesarios.(WindowsBuilder, JUnit, SVNEclipse)
- Rol Revisor Técnico:
  - PC con mouse y teclado
  - Instalación Suite de Rational 2003, para modelar los Casos de Prueba.
  - Paquete Office para la documentación de las pruebas del sistema.

2.3.4. Vista de Casos de Uso

A partir de la vista de Casos de Uso, se puede definir los escenarios o los casos de uso que serán de interés para cada iteración del ciclo de desarrollo. Esta describe los escenarios o casos de uso que tienen significación y que encapsulan la funcionalidad central del sistema.

Los Casos de Uso arquitectónicamente significativos, son aquellos que describen funcionalidades imprescindibles para el sistema, y que a través de estos se valida la arquitectura propuesta para el mismo.

- Módulo de Administración
  1. Autenticar Usuario
  2. Gestionar Roles de Usuario
  3. Gestionar Usuario
  4. Gestionar Estaciones de Trabajo

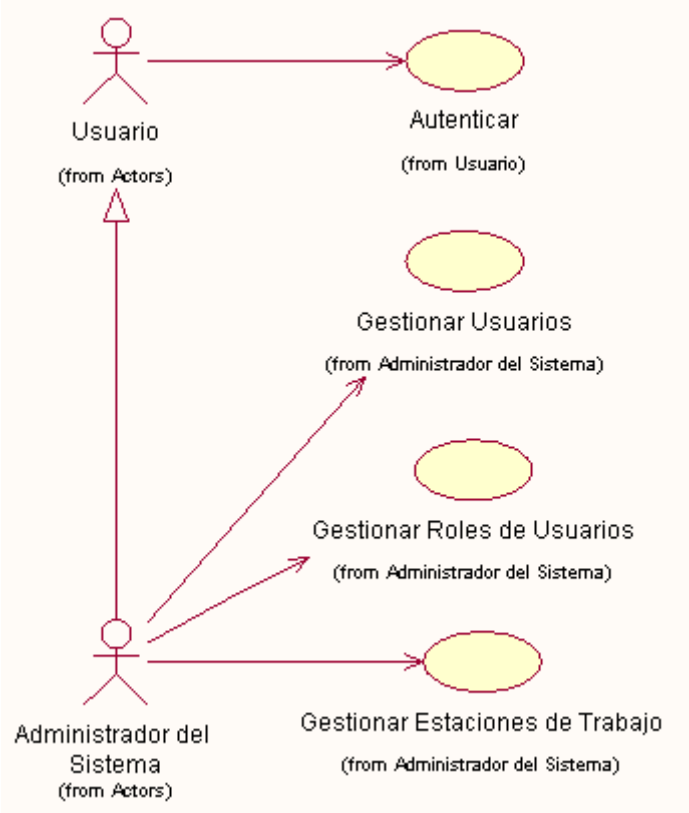
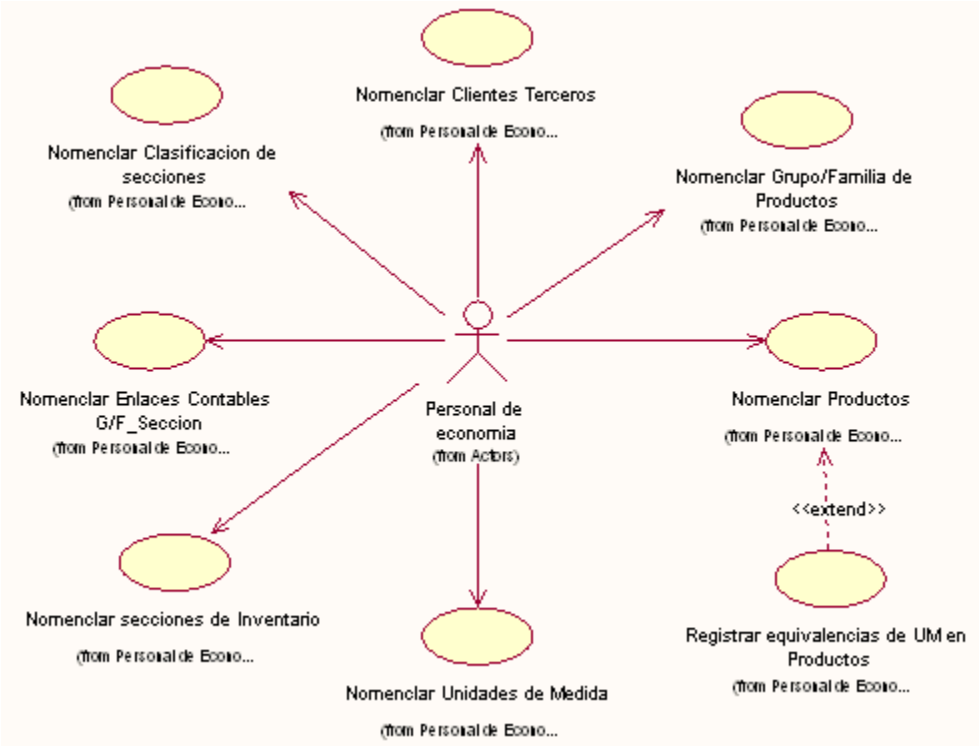


Fig. 16. Vista de caso de uso Modulo Administración

Estos Casos de Uso constituyen la base de la seguridad del sistema, a partir de ellos se puede gestionar la traza entre las operaciones, los registros de los históricos por operaciones realizadas por cada uno de los usuarios del sistema.

- Módulo Nomencladores
    1. Nomenclar Clasificación de secciones
    2. Nomenclar secciones de inventario.
    3. Nomenclar Grupo/Familia de Productos
    4. Nomenclar Unidades de Medida
    5. Nomenclar Especialidad de Proveedor
    6. Nomenclar Productos
    7. Nomenclar Unidad de Medida de Producto
    8. Registrar equivalencias de UM en Productos

9. Nomenclar Proveedor
10. Nomenclar Enlaces contables G/F_Seccion



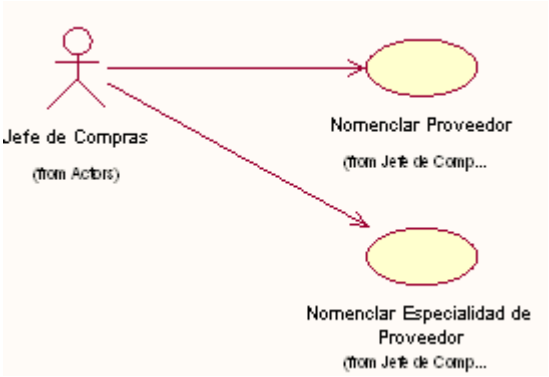


Fig. 17. Vista de caso de uso Modulo Nomencladores

Estos Casos de Uso son la base para el posterior funcionamiento del sistema, no se puede realizar ninguna de las operaciones definidas por el sistema si no se cuenta con la base conceptual para realizar las operaciones.

A partir de estas funcionalidades, se puede comprobar que la arquitectura funciona para los elementos fundamentales y básicos para su desarrollo, y el comportamiento estable de la misma.

Nomenclar Clasificación de secciones: Permite definir las clasificaciones de secciones y las características que deben cumplir.

Nomenclar secciones de inventario: Posibilita al actor el registro, modificación y eliminación de las secciones de inventario verificando las correspondientes restricciones.

Nomenclar Grupo/Familia de Productos: Posibilita al actor el registro, modificación, eliminación (si no existen dependencias) de los grupos de productos y sus respectivas familias.

Nomenclar Unidades de Medida: Permita la definición, modificación y eliminación de la unidades de medida producto con las que trabajara la entidad así como establecer equivalencias entre ellas.

Nomenclar Especialidad de Proveedor: Permite registrar, modificar y eliminar especialidades de proveedores, así como relacionarlas a los G/F correspondientes.

Nomenclar Producto: Permite registrar, modificar y eliminar (verificando dependencias) la información de los productos.

Registrar Equivalencias de unidad de medidas: Permite registrar las equivalencias entre las unidades de medida en que pueden encontrarse los productos.

Nomenclar Proveedor: Permite registrar, modificar y eliminar (verificando dependencias) la información de los proveedores.

Nomenclar Enlaces contables G/F_Seccion: Permite registrar, modificar y eliminar los enlaces contables a las cuantas de inventario (principalmente G/F-Sección-Cuentas)

- Módulo Almacén/Inventario
    1. Registrar Operación
    2. Validar Operación contable
    3. Entrar Productos por compras

4. Rebajar Productos por movimiento
5. Entrar Productos por movimiento
6. Solicitar Productos

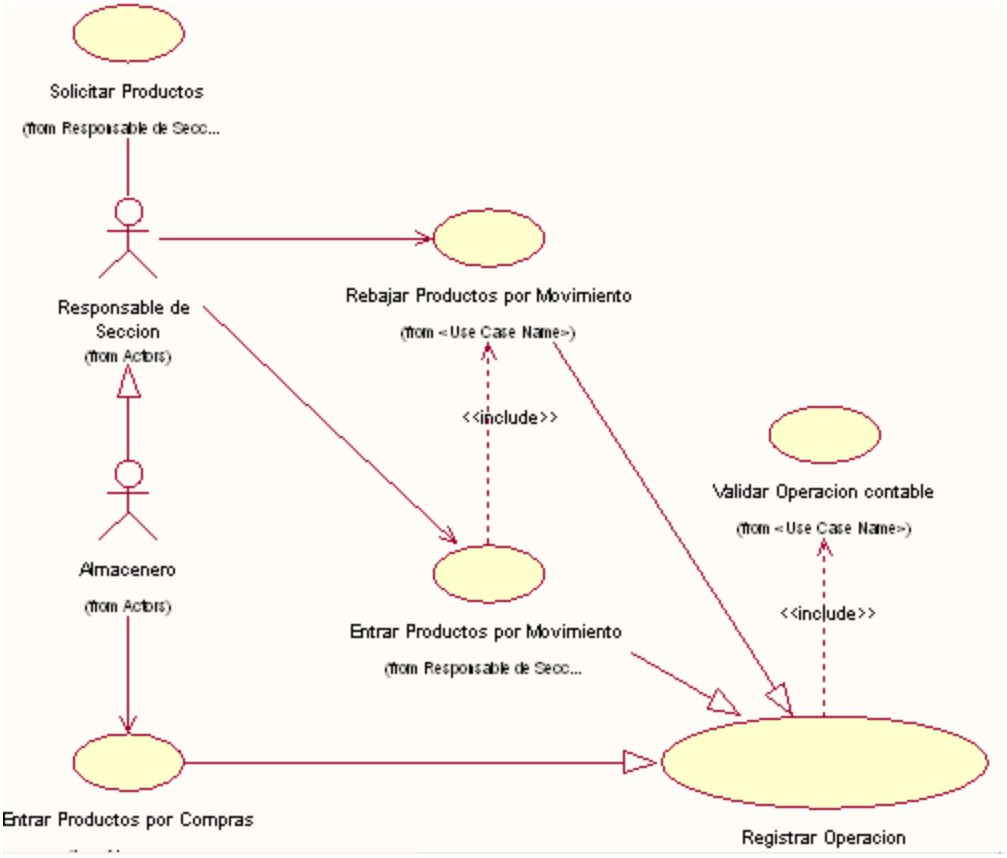


Fig. 16. Vista de caso de uso Modulo Inventario

Estos Casos de Uso encierran las funcionalidades correspondientes a las operaciones básicas que debe realizar el sistema, a partir de aquí se pueden extender las demás funcionalidades.

Registrar Operación: Permite registrar la realización de cualquier operación, actualizando las existencias de los productos afectados en las secciones involucradas, da inicio a realizar la conciliación contable (ver CU: Validar operación contable).

Validar Operación Contable: Posibilita la conciliación contable de cada una de las operaciones registradas generando además un comprobante contable según la manera de emitirlos definido por la unidad.

Entrar Productos por compras: Posibilita el registro de una compra de productos suministrados por el proveedor, avalado por el registro de la información del Informe de Recepción.

Rebajar Productos por movimientos: Posibilita el registro de una salida de productos de la sección por movimiento a otra, así como el registro de la información del Vale de Entrada.

Entrar Productos por movimientos: Permite el registro de la entrada de productos a la sección por movimiento desde otra, así como el registro de la información del Vale de Entrada.

Solicitar Productos: Permite realizar la solicitud de productos a de una sección del almacén a otra, comprobando las existencias y si se puede satisfacer la solicitud o no.

2.3.5. Vista Lógica

En la descripción de la arquitectura, la vista lógica describe las clases más importantes que formarán parte del ciclo de desarrollo. Se describe los paquetes más abstractos del sistema y las relaciones que entre ellos existen ya sea de dependencia o de uso.

La siguiente figura ilustra la división en módulos del sistema a construir, esto se hace con el objetivo de hacer el sistema más reusable y facilitar el mantenimiento ya que cualquier cambio en los procesos del negocio se tendría que cambiar solo en el módulo al que pertenece la funcionalidad que debe cambiar, facilita tambien el trabajo del equipo de desarrollo ya que permite que se puedan ir desarrollando módulos paralelamente y saber cual es la dependencia entre los mismos.

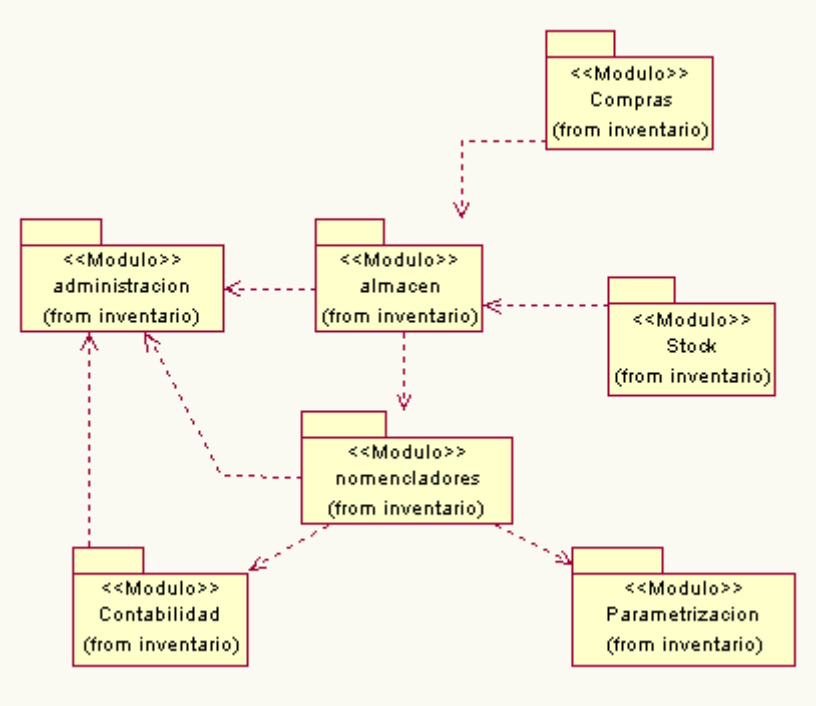


Fig. 18. Módulos del Sistema de Gestión de Inventario y Almacén

Los módulos principales identificados son:

1. **Administración**: Contiene la realización de los Casos de Uso correspondiente con la Gestión de Usuarios, Roles, Terminales.
2. **Parametrización:** Contiene la realización de los Casos de Usos correspondiente con la configuración de las estructura de la organización, la configuración de los comprobantes contables, dígitos de los códigos y otros.
3. **Nomencladores:** Contiene la realización de los Casos de Uso de nomenclar producto, sección, unidad de medida y otros.
4. **Almacén/Inventario:** Contiene la realización de los Casos de Uso de las principales operaciones que se realizan en el sistema, por ejemplo: entrar productos por compra, solicitar productos, rebajar productos por movimientos y otras.
5. **Stock:** Contiene la realización de los Casos de Uso que permiten la gestión de Stock de la organización.
6. **Compras:** Contiene la realización de los Casos de Uso que permitan la gestión de las compras y los procesos que de ella se derivan.

7. **Contabilidad:** Contiene la realización de los Casos de Uso que permita configurar los parámetros fundamentales para el que sistema pueda realizar operaciones como la Validación Contable, y la generación de los comprobantes contables.

La distribución de la aplicación para cada uno de los módulos se hará de la siguiente forma:

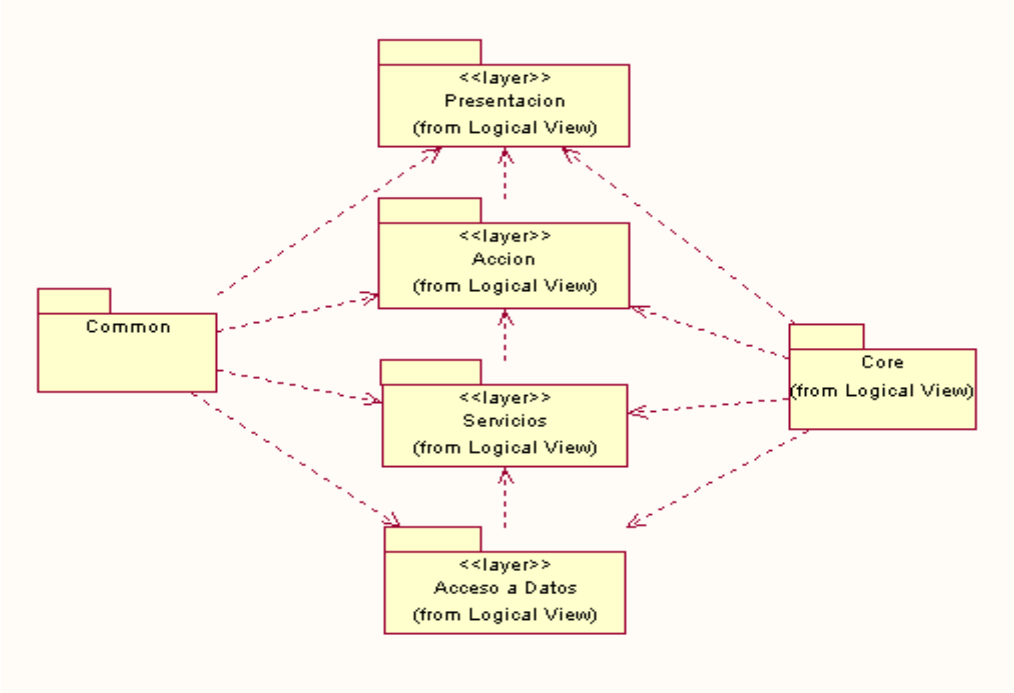


Fig. 19. Estructura en capas del sistema

- **Presentación**: Clases y componentes del framework Swing.
- **Acción**: Clases que controlan las acciones de la presentación.
- **Servicios**: Clases que controlan la Lógica del Negocio, componentes del framework Spring
- **Acceso a Datos** : Clases que controlan la lógica transaccional y la interacción con la Base de Datos, componentes del framework Hibernate.

- **Common:** Objetos de Negocio proporcionados por el mapeo de las clases del framework Hibernate, como estos no guardan información de estado que los identifican como objetos de persistencia se pueden utilizar desde cualquier lado de la aplicación.
- **Core:** Paquete que encierra clases comunes, componentes comunes para todas las capas de la aplicación, contiene la configuración de los frameworks.

Los principales subsistemas y las interfaces que permiten la comunicación entre ellos son:

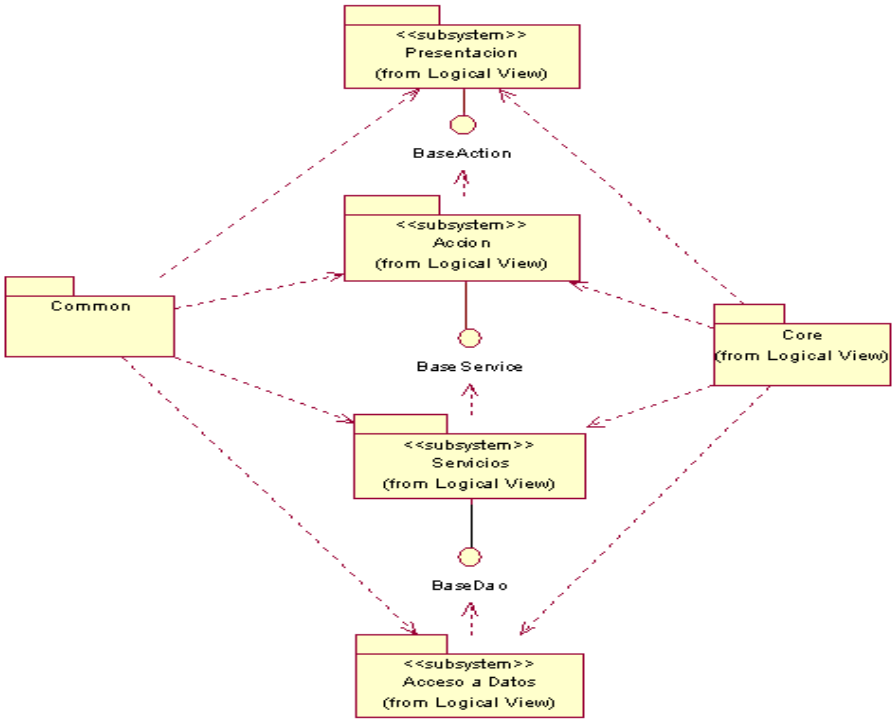


Fig. 19. Subsistemas de diseño de la aplicación

Las implementaciones de las Interfaces BaseAction, BaseService y BaseDao pueden verse en la sección 2.2.3 Organigrama de la arquitectura.

El paquete **Core** contiene los siguientes paquetes:

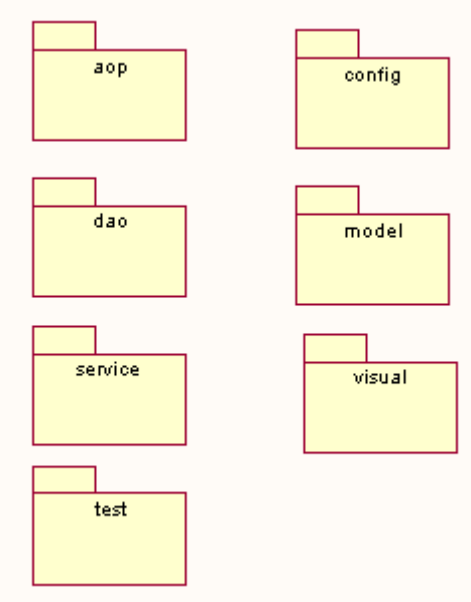


Fig. 20. Paquetes Core

- El paquete **aop** contiene la configuración del framework Spring, los *aspectos* y propiedades.
- El paquete **dao** contiene la declaración y la implementación de la interfaz BaseDao.
- El paquete **service** contiene la declaración y la implementación de la interfaz BaseService.
- El paquete **model** contiene la implementación de la clase BaseObject y la configuración del framework de Hibernate.
- El paquete **test** contiene la declaración y la implementación de la Interfaz para usar en las pruebas de JUnit BaseTestCase.
- El paquete **config** contiene la implementacion del patrón Service Locator y de BaseBean.

El paquete **common** contiene las clases Objetos del mapeo, los ficheros XML del mapeo y los ficheros para las consultas (query).

La estructura de las capas de Presentación y Acciones es la siguiente:

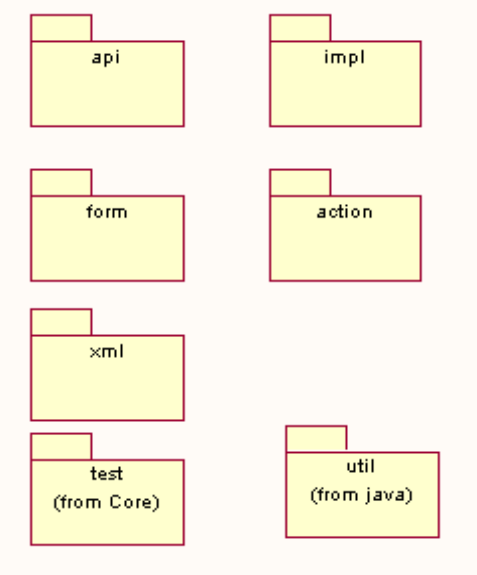


Fig. 21. Paquetes Capa Presentación y Acción

- Paquete **api:** Contiene las interfaces que hagan falta para la implementación de alguna funcionalidad.
- Paquete **impl:** Contiene las implementaciones que se quieran hacer de las clases interfaces declaradas.
- Paquete **form:** Contiene las clases que de interfaz de usuario que permita al usuario interactuar con la aplicación.
- Paquete **action:** Contiene las clases Action para cada una de las formas, en caso de que BaseAction no resuelva esa funcionalidad.
- Paquete **util:** Contiene las clases que pueden ser útiles para la capa.

La estructura de las capa de Servicios es la siguiente:

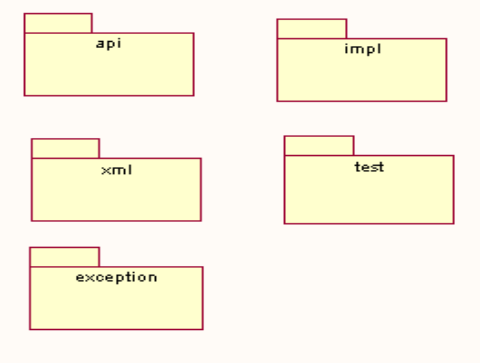


Fig. 22. Paquetes Capa de Servicio

- Paquete **api:** Contiene las interfaces de los servicios que hagan falta para la implementación de alguna funcionalidad.
- Paquete **impl:** Contiene las implementaciones que se quieran hacer de las clases interfaces declaradas.
- Paquete **xml:** Contiene los ficheros XML para la configuraci'on de los Beans de cada uno de los servicios.
- Paquete **test:** Contiene las pruebas de unidad (JUnit) para cada uno de los servicios implementados.
- Paquete **exception:** Contiene las clases para el manejo de las excepciones.

La estructura de las capas de Acceso a Datos es la siguiente:

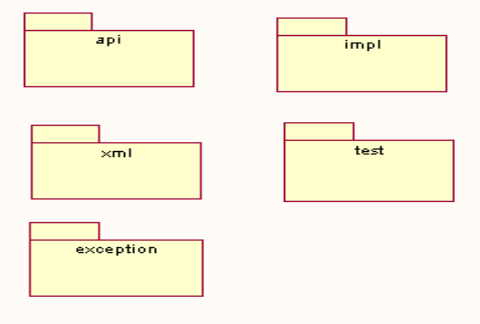


Fig. 23. Paquetes Capa de Acceso a Datos

- Paquete **api:** Contiene las interfaces de los servicios que hagan falta para la implementación de alguna funcionalidad.
- Paquete **impl:** Contiene las implementaciones que se quieran hacer de las clases interfaces declaradas.
- Paquete **xml:** Contiene los ficheros XML para la configuraci'on de los Beans de cada uno de los servicios.
- Paquete **test:** Contiene las pruebas de unidad (JUnit) para cada uno de los servicios implementados.
- Paquete **exception:** Contiene las clases para el manejo de las excepciones.

2.3.6. *Vista Implementación*

La vista de implementación proporciona una descripción de las principales capas y subsistemas de componentes de la aplicación. Los paquetes principales de la aplicación por cada uno de los módulos son:

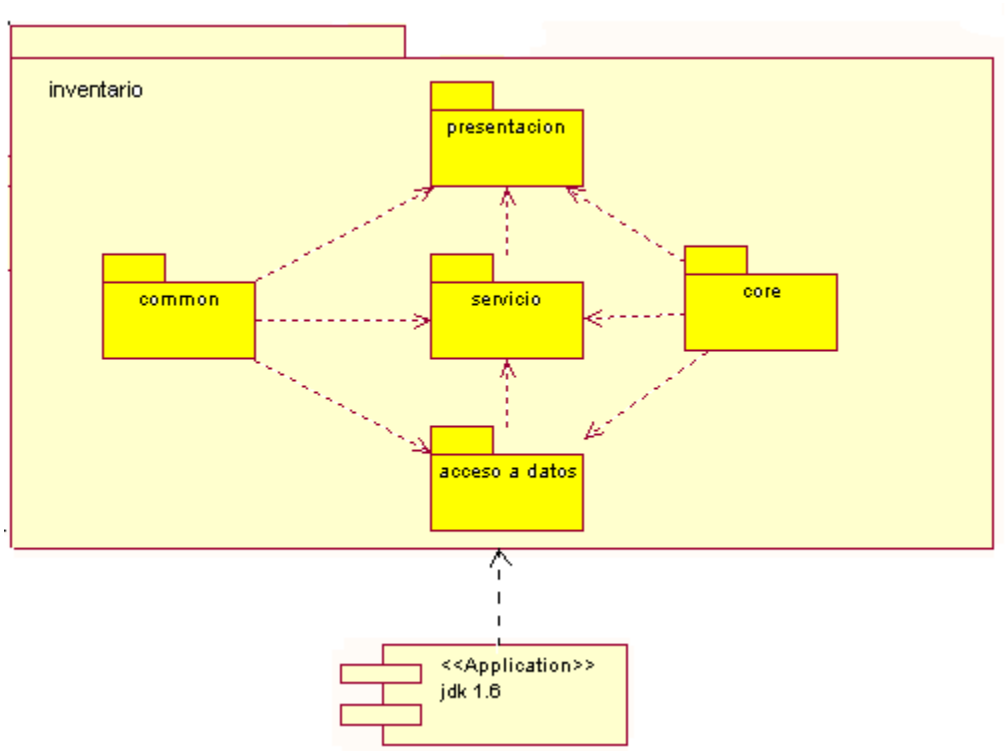


Fig. 24. Vista general del modelo de implementación

Cada uno de dichos paquetes encapsulan uno o más componentes que se interrelacionan entre ellos para darle solución a la aplicación y todos depende de la JDK que debe de estar instalada en la estación de trabajo que decida ser instalada la aplicación.

Los componentes están distribuidos en las capas y paquetes de la aplicación de la siguiente forma:

- Core

Contiene los componentes fundamentales para el trabajo en cada una de las capas:

1. **BaseForm**: Componente del cual heredan las Forms de la aplicación.

2. **BaseInternalForm**: Componente del cual van a heredar todas los InternalForm de Swing creados en la aplicación.

3. **BaseBean**: Componente que permite que todos los Bean creados puedan utilizar el ServiceLocator.

4. **BaseAction**: Componente que contiene las funcionalidades básicas de las acciones para todas las Forms de la aplicación.

5. **BaseService**: Componente que contiene las funcionalidades básicas de todos los servicios de la aplicación.

6. **BaseDao**: Componente que contiene las funcionalidades básicas que maneja el acceso a los datos de la aplicación.

7. **FormNames**: Componente que maneja cada una de las WindowForm creadas en la aplicación.

8. **ActionNames**: Componente que maneja cada una de las clases Action de la subcapa de Acciones creadas en la aplicación.

9. **ServiceNames**: Componente que maneja cada uno los Servicios de la aplicación.

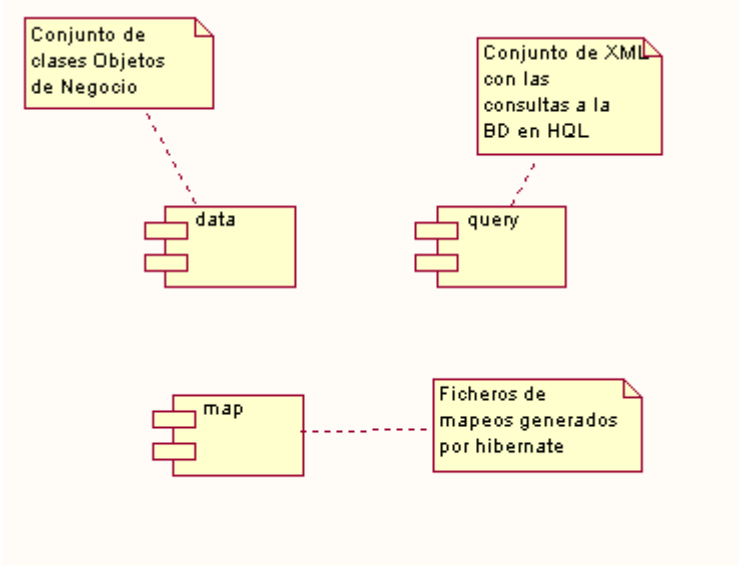
10. **DaoNames**: Componente que maneja cada uno de los DAO (Objetos de Acceso a Datos) creados en la aplicación.

11. **Paquete init**: Conjunto de componentes que manejan el arranque de la aplicación, la configuración del contexto de la aplicación.

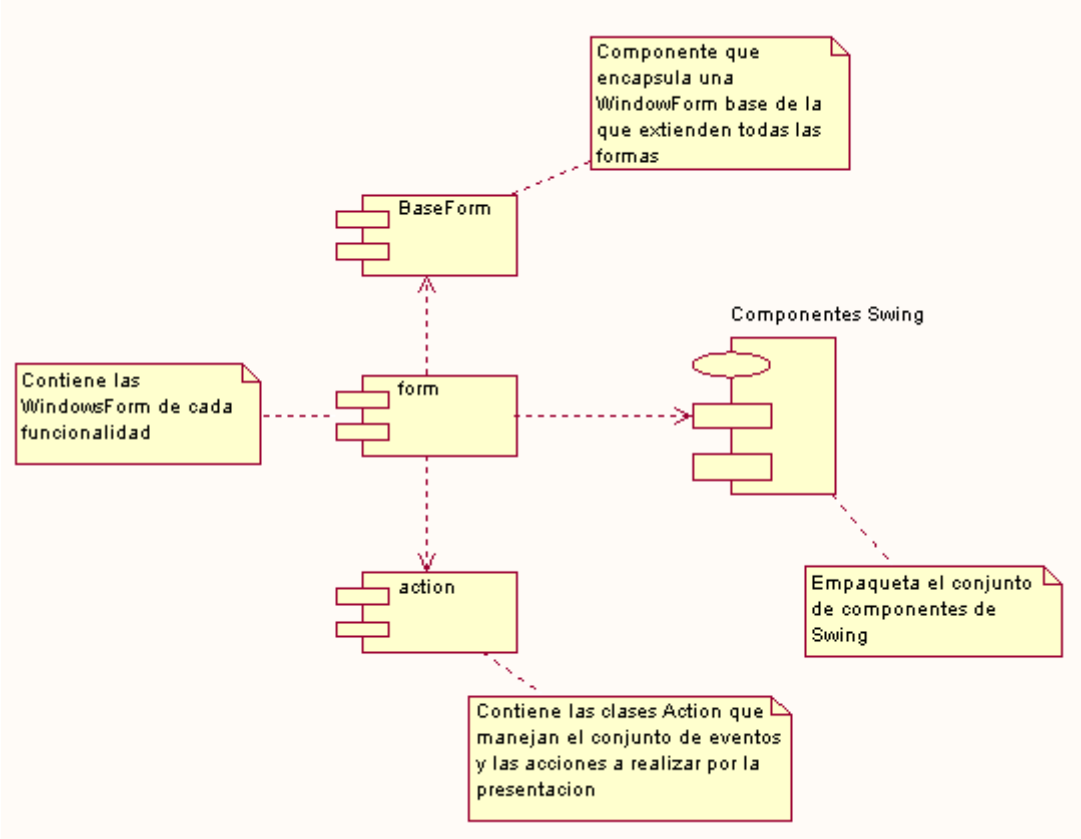
12. **Paquete VisualUtil**: Contiene componentes de utilidad para las distintas capas por ejemplo: ColumnNames y SearchParameters.

13. **SearchParameter:** Contiene los componentes que permiten obtener los parámetros de búsquedas para la capa de presentación

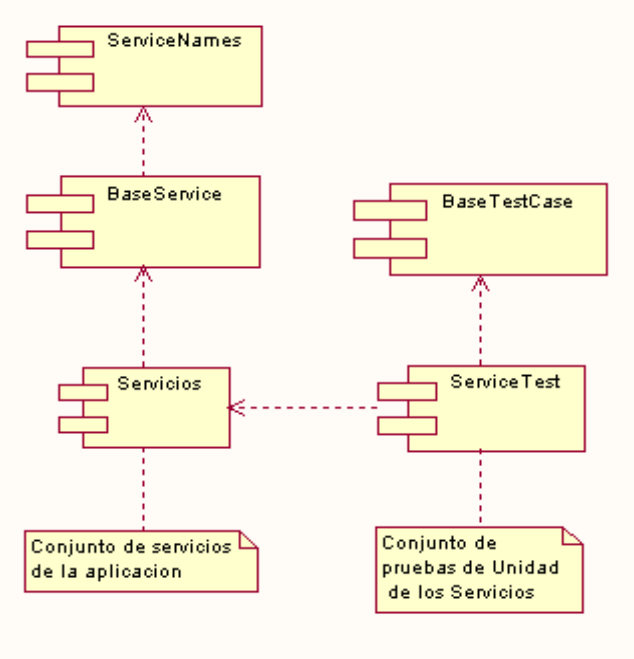
- Common



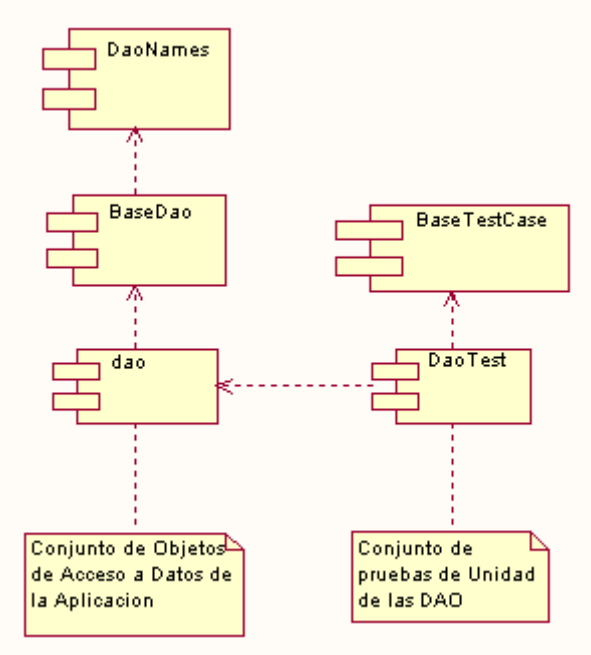
- Presentación



- Servicios



- Acceso a Datos



El conjunto de paquetes identificados permiten que cualquiera de ellos pueda ser reutilizado, en esta y en futuras aplicaciones, permite además el mantenimiento de la aplicación.

2.3.7. Vista Despliegue

La vista de despligue propone la distribución física del sistema y como estarán distribuidos los componentes de la aplicación en ellos, y como se satisfacen los requisitos no funcionales de software y hardware e influye en el rendimiento.

- Nodo **PC Servidor:** Constituye el nodo servidor, es importante tener presente que en el se encuentra el servidor de Base de Datos, ya que no existe un servidor de aplicación, la base de datos es centralizada asi que todas las PC clientes deben poder conectarse al mismo servidor de aplicación. Contiene el ejecutable de la aplicación para la configuración inicial de la misma, aunque esto no es un requisito indispensable, la misma se puede configurar desde una PC Cliente.

- Nodo **PC Cliente**: Contiene el ejecutable de la aplicación y la Java Virtual Machine (Maquina Virtual de Java). La utilización de los recursos de la PC Cliente viene dado por los módulos de la aplicación que el cliente decida instalar, dado por el desarrollo en módulo de la aplicación, así se evita sobrecargar las PC Clientes con elementos de la aplicación que no se serán usados.

- Los dispositivos de scanner e impresora satisfacen los requisitos de los clientes de impresión de reportes generados por la aplicación, y la gestión de los proveedores y sus contratos posibles a scannearse. Los mismos no se le especifican la forma de conectarse ya que pueden ser por puerto **USB** o en **Paralelo** lo que no es significativo para la aplicación aunque influye en la rapidez de la impresión pero esto no constituye una exigencia del cliente.

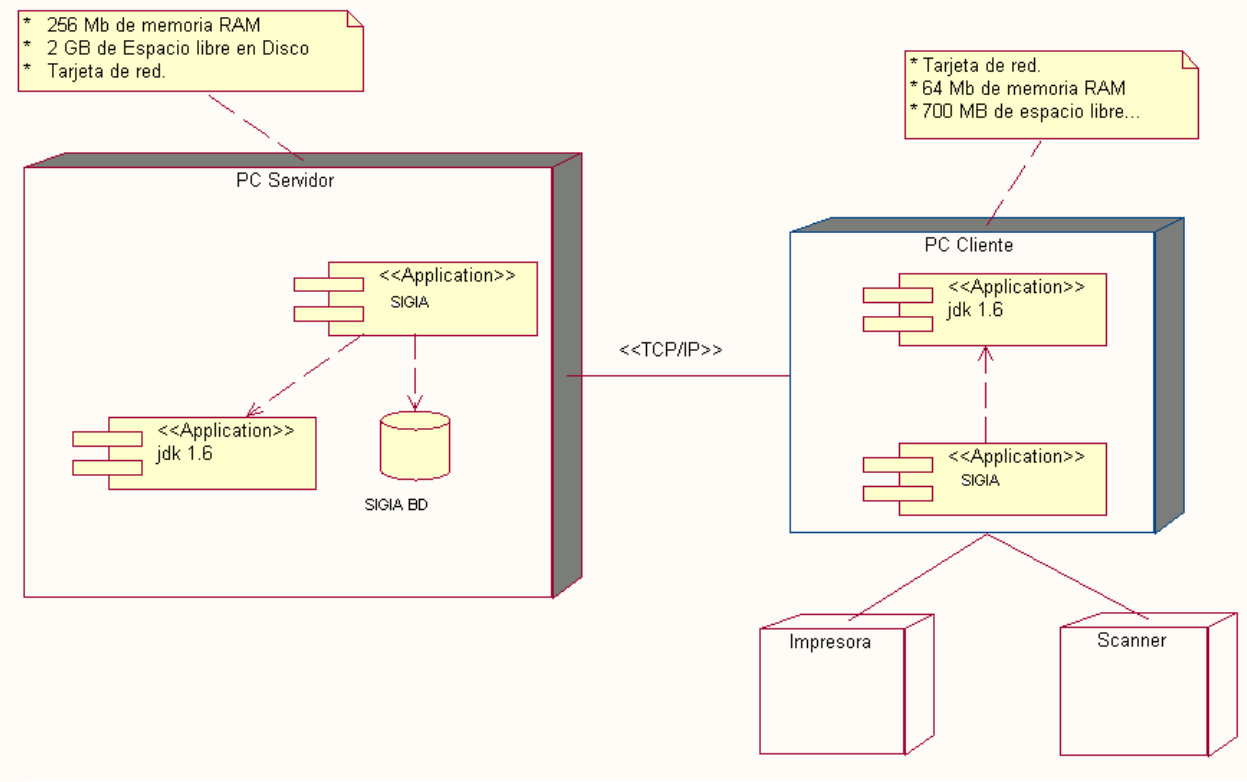


Fig. 25. Modelo de despliegue

No es necesario especificar otra variante para el modelo de despliegue ya que la aplicación sigue una arquitectura Cliente /Servidor en su variante Cliente enriquecido, así que la lógica se concentra en los clientes, reservando el servidor solo para la Base de Datos

2.4. Valoración final de la solución propuesta

La solución propuesta parte de la presentación de dos documentos que en conjunto conforman la descripción arquitectónica del Sistema de Gestión de Inventario y Almacén.

La utilización del documento Línea Base de la Arquitectura constituye un apoyo al documento Descripción de la Arquitectura propuesta por la metodología de desarrollo RUP (Proceso Unificado de Software), el ubicar toda la información que se recoge en la Línea Base en el mismo documento de descripción, haría de este último un documento muy denso y que no pudiera recoger toda la información y la especificación que tiene de esta forma.

La organización de la Línea Base está constituida por los aspectos que se ha entendido son los más importantes para la descripción, el organigrama de la arquitectura explica detalladamente el estilo de arquitectura propuesto, sus principales componentes, conectores y sus configuraciones, las restricciones impuestas por el estilo solo se aborda la más importante, aunque en el tiempo de desarrollo se deberá refinar en el caso de que haya que imponer alguna otra restricción. El estilo de Arquitectura en Capas permite la especialización de los programadores en una parte de la implementación del sistema pero tiene como desventaja que crea una dependencia entre los desarrolladores, pudiendo ocasionar retrasos en las capas por dependencias con otras. Con este estilo se propicia la reutilización de componentes para cada uno de los módulos y se hace un mejor manejo de los errores ya que es facil identificar en que capa se encuentra el mismo.

Uno de los aspectos más importantes son la utilización de los patrones de arquitectura que se deberán poner en práctica en la aplicación, es necesario la profundización entre los términos Estilos de Arquitectura, Patrones de Arquitectura y Patrones de Diseño. En la descripción arquitectónica no se propone la utilización de ningún patrón de diseño, ya que los mismos deberán surgir a partir de problemas en el diseño de clases de la aplicación y quedaría en un nivel muy bajo de la abstracción quedando fuera de las actividades del arquitecto, aunque queda bien claro que el arquitecto es responsable que el diseño de clases cumpla con los aspectos del diseño arquitectónico.

Se ha expuesto los patrones arquitectónicos fundamentales que se utilizarán en cada una de las capas de la aplicación, esto también se deberá estar revisando constantemente durante el desarrollo del sistema, ya que pudiera sufrir cambios o adecuaciones que se deben ir refinando.

Otro aspecto importante es la definición de la tecnología y las herramientas que se utilizarán en la implementación del sistema, la única reserva en cuanto este aspecto puede ser que el disparo de los costes del desarrollo del proyecto, al ser la tecnología propuesta demandante de recursos de hardware para la implementación, aspecto este a analizar directamente con los clientes.

El documento Descripción de la Arquitectura propone la utilización de la 4 de las vistas propuestas por la metodología de desarrollo RUP (Proceso Unificado de Desarrollo),la no

utilización de la Vista de Procesos puede dejar de especificar algún proceso que sea concurrente, aunque hasta el momento en el que se ha especificado el sistema no se ha encontrado ninguno, sería bueno el análisis de cada uno de los escenarios especificados para la revisión de si se necesita dicha vista y que aportaría a la descripción del sistema.

Las principales clases del diseño están propuestas desde la Línea Base, la utilización de estas tres interfaces, BaseAction, BaseService, BaseDAO, contiene un conjunto de métodos que permiten ejecutar cualquier transacción dentro de la aplicación, solo es necesario la aplicación de los métodos que contengan la lógica y la aplicación de las distintas reglas de negocio para cada una de las operaciones que se vayan a realizar.

Otro aspecto a resaltar es la utilización de los distintos frameworks de desarrollo, los que posibilitan un desarrollo rápido de la aplicación ya que contiene un conjunto de componentes bien especificados, y con sus interfaces bien definidas, el mayor esfuerzo en la utilización de estos framework se concentra en la configuración del mismo, ya que su uso se hace muy fácil.

Atendiendo a todos estos aspectos se considera que la arquitectura propuesta responde a los principales puntos a medir, la misma satisface los requisitos de los clientes, es construible por parte de los desarrolladores, puede ser probada y es administrable por los arquitectos y el jefe de proyecto. A partir de estos señalamientos se propone un conjunto de recomendaciones.

2.5. Conclusiones

La propuesta de solución dada en este capítulo abarca dos documentos de obligada construcción, y constante refinamiento por parte del arquitecto de software del sistema.

CONCLUSIONES

- Se hizo un estudio detallado de los principales aspectos de la descripción arquitectónica.
- Se realizó una valoración de las principales tareas que debe llevar a cabo el rol de arquitecto según la metodología de desarrollo RUP (Proceso Unificado de Desarrollo).
- Se definió los principales estilos arquitectónicos, sus componentes, configuraciones y restricciones, impuestas por los requisitos de los clientes.
- Se realizó un estudio detallado de las principales categorías de patrones arquitónicos, se fundamentó la elección de los mismos para su aplicación en el sistema y se brindó la solución al mismo.
- Dada las restricciones impuestas por algunos requisitos no funcionales del cliente y en función de los principales atributos de calidad de la Arquitectura de Software se definieron las principales tecnología y herramientas a utilizar en el desarrollo del sistema, y se configuró cada uno de los puestos de trabajo por roles en el equipo de desarrollo.
- Se cumplió con cada uno de los aspectos de la descripción de la arquitectura propuesta por la metodología de desarrollo, y se propuso la implementación de componentes de software para facilitar la reutilización de los mismos.
- Con la Arquitectura de Software propuesta cumple con: satisface los requisitos de los clientes, es “construible” por los desarrolladores, es “verificable” puede ser probada por los revisores y es “administrable” por los administradores del sistema.

RECOMENDACIONES

- El refinamiento constante de la arquitectura propuesta, durante el ciclo de desarrollo.
- La revisión de cada uno de los escenarios de la aplicación para evaluar a fondo las restricciones que se le imponen a la arquitectura.
- El análisis de las dependencias entre los subsistemas de implementación para evitar que existan cuellos de botella en alguno de ellos y que se vea afectado en cuanto a tiempo la entrega de cada una de las actividades de implementación.
- Realizar una evaluación de los costes de la tecnología utilizada, aunque la misma en su totalidad es libre, es necesario evaluar los costes del hardware que la soportan.
- La aplicación de cualquiera de los métodos de evaluación de la arquitectura, con el objetivo de identificar las principales debilidades, tanto en la descripción arquitectónica como en el propio diseño arquitectónico.

REFERENCIAS BIBLIOGRAFICAS

[Colectivo Autores, 1999] Autores, C. d. (1999) Guía de Iniciación al Lenguaje JAVA. Junta de Castilla y Leon, Url: http://pisuerga.inf.ubu.es/lsi/Invest/Java/Tuto/I_3.htm

[Bass, Kazman 1998] Bass, L., Clements, P., y Kazman, R. (1998) Software Architecture in Practice. Addison Wesley.

[Bastarrica, 2005] Bastarrica, M. C. (2005) Atributos de Calidad y Arquitectura del Software. Volume, 4

[Bauer, 2005] Bauer, C. (2005). Hibernate in Action. A Guide to the concepts and practices of object/relational mapping. M. Publications.Co.

[Booch, 1999] Booch, G. (1999). The Unified Modeling Language User Guide, Addison Wesley.

[Bosch, 2000] Bosch., J. (2000). Design & Use of Software Architectures". Addison Wesley.

[Box,1998] Box, D. (1998). Essential COM, Addison Wesley.

[Brown et al, 1998] Brown, A. W. y. W., K. C. (1998) The Current State of CBSE. IEEE Software

[Reynoso, 2004] Carlos Reynoso, N. K. (2004) Estilos y Patrones en la Estrategia de Arquitectura de Microsoft. UNIVERSIDAD DE BUENOS AIRES

[CCITT/ITU-T, 1998] CCITT/ITU-T (1988). Redes de Comunicacion de Datos. Interconexion de Sistemas Abiertos: Modelo y Notacion.

[Clements, 1996] Clements, P. (1996). A Survey of Architecture Description Languages. Proceedings of the International Workshop on Software Specification and Design.

[Deepak, 2001] Deepak Alur, J. C. a. D. M. (2001). Core J2EE Patterns: Best Practices and Design Strategies. P. H. S. M. Press.

[Perry, Wolf, 1992] Dewayne E. Perry, A. L. W. (1992). "Foundations for the study of software architecture."

[Fayad et al, 1999]Fayad, M., Schmidt, D., y Johnson, R. E. (1999). Application Frameworks., Wiley & Sons. 3.

[Eclipse Foundation, 2007] Foundation, E. (2007) Eclipse (software). Wikipedia

[Gamma, 2002] Gamma, E. (2002). Patrones de Diseño, Addison-Wesley.

[Gómez, 2006] Gómez, S. P. (2006) Swing Univ. Politécnica de Madrid

[Astudillo, 1998] Hernán Astudillo, S. H. (1998) Understanding the Architect's Job. Software Architecture Workshop of OOPSLA'98

[IEEE-1471, 2000] IEEE-1471 (2000) Recommended Practice for. Architectural Description of Software-Intensive Systems

[IEEE Std 1061-1992, 1992]IEEE (1992) Standard for a software quality metrics methodology. IEEE Std 1061-1992

[ISO/ITU-T, 1995] ISO/ITU-T (1995) Reference Model for Open Distributed Processing. International Organization for Standarization.

[Jacopbson, 1999] Jacopbson, I. (1999). The Unified Software Development Process., Addison Wesley.

[Johnson, 1997] Johnson, E. (1997) Frameworks = Components + Patterns. Communications of the ACM.

[Kruchten, 1995] Kruchten, P. (1995). The 4+1 View Model of Architecture., IEEE Software.

[Bass et al, 2003] Len Bass, P. C., Rick Kazman (2003). Software Architecture in Practice (2nd Ed.), Addison Wesley.

[Luckham, 1995] Luckham, D. C. (1995) Specification and Analysis of System Architecture using Rapide. IEEE Trans. on Software Engineering.

[Bertoa et al,]Manuel F. Bertoa, J. M. T. y. A. V. Aspectos de Calidad en el Desarrollo de Software Basado en Componentes.

[Shaw, 1996] Mary Shaw, D. G. (1996). Software Architecture. Perspectives on an Emerging Discipline, Prentice Hall.

[Medvidovic et al, 1997] Medvidovic, N. y. R., D. S. (1997) Domains of Concern in Software Architectures and Architecture Description Languages. En USENIX Conf. on Domain-Specific Languages, Santa Barbara (Estados Unidos).

[Microsoft, 1996] Microsoft (1996) DCOM - The Distributed Component Object Model.

[Sun Microsystem, 2005] Microsystem, S. (2005). "The Java(tm) Virtual Machine Specification."

[Sun Microsystem, 1997] Microsystems, S. (1997) JavaBeans API Specification 1.01.

[OMG, 1999] OMG (1999) The Common Object Request Broker: Architecture and Specification.

[OSF, 1994] OSF (1994) OSF DCE Application Development Guide. Open Software Foundation, Cambridge, MA (Estados Unidos).

[Palos, 2006] Juan Antonio Palos (2006) Catálogo de Patrones de Diseño J2EE. Y II: Capas de Negocio y de Integración, Sun Microsystem, Url: http://www.programacion.net/java/tutorial/patrones2/8/

[Monroe et al, 1997] Robert Monroe, A. K., Ralph Melton y David Garlan (1997) Architectural Styles, design patterns, and objects. IEEE Software.

[Rogerson, 1997] Rogerson, D. (1997). Inside COM. M. Press.

[Rumbaugh, 1991] Rumbaugh, J. (1991). Object-Oriented Modeling and Design, Prentice Hall.

[Rumbaugh, 1999] Rumbaugh, J. (1999). The Unified Modeling Language Reference Manual, Addison Wesley.

[SEI, 1990] SEI (1990) Workshop on Domain-Specific Software Architectures. Software Engineering Institute.

[Shaw, Garlan, 1996] Shaw, M. y. G., D. (1996). Software Architecture. Perspectives of an Emerging Discipline, Prentice Hall.

[Sommers, 2006] Sommers, F. (2006) A Framework for Swing. Artimar Developer

[Banerjee et al, 2005]Somo Banerjee, C. A. M., Nenad Medvidovic, Leana Golubchik "Leveraging Architectural Models to Inject Trust into Software Systems." 7.

[Szyperski, 2000] Szyperski, C. (2000). Components and Architecture. Software Development Online: 10.

[UPV/EHU, 2006] UPV/EHU (2006). Arquitecturas Software de varios niveles en Java.

[Velazco, 2000] Velazco, C. C. (2000). Un lenguaje para la Especificación y Validación de Arquitecturas de Software. Departamento de lenguajes y Ciencias de la Computación. Málaga, Universidad de Málaga: 266.

[Venners, 2000] Venners, B. (2000). Inside the Java Virtual Machine

[Vinoski, 1998] Vinoski, S. (1998) New Features for CORBA 3.0. Communications of the ACM

[Welicki, 2007] Welicki, L. (2007) Patrones y Antipatrones: una Introducción

[Trujillo, 2006] Yaimí Trujillo Casañola (2006) Evaluación teórica de la adopción del enfoque de Factorías de Software en la Universidad de las Ciencias Informáticas. Universidad de las Ciencias Informáticas.

BIBLIOGRAFIA

S. E. I. (2000) Software Architecture.

Chile, U. d. (2004) La Máquina Virtual de Java en el corazón del chip picoJava.

Rolando Alfredo, S. C. (2002). EL PARADIGMA CUANTITATIVO DE LA INVESTIGACIÓN CIENTIFICA. Ciudad de la Habana.

Society, S. E. S. C. o. t. I. C. (2000). "IEEE Recommended Practice for Architectural Description of Software-Intensive Systems."

[Colectivo Autores, 1999] Autores, C. d. (1999) Guía de Iniciación al Lenguaje JAVA. Junta de Castilla y Leon

Bass, L., Clements, P., y Kazman, R. (1998) Software Architecture in Practice. Addison Wesley.

Bastarrica, M. C. (2005) Atributos de Calidad y Arquitectura del Software. Volume, 4

Bauer, C. (2005). Hibernate in Action. A Guide to the concepts and practices of object/relational mapping. M. Publications.Co.

Booch, G. (1999). The Unified Modeling Language User Guide, Addison Wesley.

Bosch., J. (2000). Design & Use of Software Architectures". Addison Wesley.

Box, D. (1998). Essential COM, Addison Wesley.

Brown, A. W. y. W., K. C. (1998) The Current State of CBSE. IEEE Software

Carlos Reynoso, N. K. (2004) Estilos y Patrones en la Estrategia de Arquitectura de Microsoft. UNIVERSIDAD DE BUENOS AIRES

CCITT/ITU-T (1988). Redes de Comunicacion de Datos. Interconexion de Sistemas Abiertos: Modelo y Notacion.

Clements, P. (1996). A Survey of Architecture Description Languages. Proceedings of the International Workshop on Software Specification and Design.

Deepak Alur, J. C. a. D. M. (2001). Core J2EE Patterns: Best Practices and Design Strategies. P. H. S. M. Press.

Dewayne E. Perry, A. L. W. (1992). "Foundations for the study of software architecture."

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

Foundation, E. (2007) Eclipse (software). Wikipedia

Gamma, E. (2002). Patrones de Diseño, Addison-Wesley.

Gómez, S. P. (2006) Swing Univ. Politécnica de Madrid

Hernán Astudillo, S. H. (1998) Understanding the Architect's Job. Software Architecture Workshop of OOPSLA'98

IEEE-1471 (2000) Recommended Practice for. Architectural Description of Software-Intensive Systems

IEEE (1992) Standard for a software quality metrics methodology. IEEE Std 1061-1992

ISO/ITU-T (1995) Reference Model for Open Distributed Processing. International Organization for Standarization.

Jacopbson, I. (1999). The Unified Software Development Process., Addison Wesley.

Johnson, E. (1997) Frameworks = Components + Patterns. Communications of the ACM.

Kruchten, P. (1995). The 4+1 View Model of Architecture., IEEE Software.

Len Bass, P. C., Rick Kazman (2003). Software Architecture in Practice (2nd Ed.), Addison Wesley.

Luckham, D. C. (1995) Specification and Analysis of System Architecture using Rapide. IEEE Trans. on Software Engineering.

Manuel F. Bertoa, J. M. T. y. A. V. Aspectos de Calidad en el Desarrollo de Software Basado en Componentes.

Mary Shaw, D. G. (1996). Software Architecture. Perspectives on an Emerging Discipline, Prentice Hall.

Medvidovic, N. y. R., D. S. (1997) Domains of Concern in Software Architectures and Architecture Description Languages. En USENIX Conf. on Domain-Specific Languages, Santa Barbara (Estados Unidos).

Microsoft (1996) DCOM - The Distributed Component Object Model.

Microsystem, S. (2005). "The Java(tm) Virtual Machine Specification."

Microsystems, S. (1997) JavaBeans API Specification 1.01.

OMG (1999) The Common Object Request Broker: Architecture and Specification.

OSF (1994) OSF DCE Application Development Guide. Open Software Foundation, Cambridge, MA (Estados Unidos).

Robert Monroe, A. K., Ralph Melton y David Garlan (1997) Architectural Styles, design patterns, and objects. IEEE Software.

Rogerson, D. (1997). Inside COM. M. Press.

Rumbaugh, J. (1991). Object-Oriented Modeling and Design, Prentice Hall.

Rumbaugh, J. (1999). The Unified Modeling Language Reference Manual, Addison Wesley.

SEI (1990) Workshop on Domain-Specific Software Architectures. Software Engineering Institute.

Shaw, M. y. G., D. (1996). Software Architecture. Perspectives of an Emerging Discipline, Prentice Hall.

Sommers, F. (2006) A Framework for Swing. Artimar Developer

Somo Banerjee, C. A. M., Nenad Medvidovic, Leana Golubchik "Leveraging Architectural Models to Inject Trust into Software Systems." 7.

Szyperski, C. (2000). Components and Architecture. Software Development Online: 10.

[UPV/EHU, 2006] UPV/EHU (2006). Arquitecturas Software de varios niveles en Java.

Velazco, C. C. (2000). Un lenguaje para la Especificación y Validación de Arquitecturas de Software. Departamento de lenguajes y Ciencias de la Computación. Málaga, Universidad de Málaga: 266.

Venners, B. (2000). Inside the Java Virtual Machine

[Vinoski, 1998] Vinoski, S. (1998) New Features for CORBA 3.0. Communications of the ACM

Welicki, L. (2007) Patrones y Antipatrones: una Introducción

Yaimí Trujillo Casañola (2006) Evaluación teórica de la adopción del enfoque de Factorías de Software en la Universidad de las Ciencias Informáticas. Universidad de las Ciencias Informáticas.