# Arquitectura de Sistemas de Información basados en Componentes sobre la Plataforma J2EE

Daniel Perovich – Leonardo Rodríguez – Andrés Vignaga
{perovich, lrodrigu, avignaga}@fing.edu.uy

Instituto de Computación – Facultad de Ingeniería – Universidad de la República
Julio Herrera y Reissig 565 5to. piso, Montevideo 11300, Uruguay
Fax: +598–2–711 0469

http://www.fing.edu.uy/inco

## Resumen

El desarrollo de sistemas basado en componentes puede ser atacado desde dos frentes o más precisamente niveles diferentes. Uno de ellos es el nivel de la tecnología que se empleará para la implementación del sistema, y el otro es un nivel previo más abstracto en el que el centro es la estructura lógica de la solución dejando de lado aspectos particulares de la tecnología. El enfoque de Model Driven Architecture incorpora esta separación distinguiendo modelos de sistemas que son independientes de la plataforma de desarrollo de los que son específicos para la misma. Alineado con este enfoque, este artículo propone una correspondencia entre la arquitectura lógica en capas de un sistema de información creado independiente de la tecnología aplicando una propuesta metodológica conocida, y las construcciones de la plataforma J2EE. Esta correspondencia o mapping permite definir transformaciones entre modelos independientes de la plataforma resultantes de la aplicación de la metodología mencionada que permiten un razonamiento abstracto de la solución, y modelos específicos de la plataforma que se encuentran alineados con las construcciones de la tecnología y que son implementables en forma directa.

**Palabras clave:** Arquitectura de software, Desarrollo basado en componentes, Sistemas de información, Java 2 Enterprise Edition, Enterprise Java Beans, Model Driven Architecture

## 1. Introducción

El desarrollo basado en componentes (CDB) se encuentra en permanente evolución. En particular, dicha evolución se experimenta en dos mundos diferentes. Por un lado, puede encontrarse estudios del CDB a nivel conceptual y/o metodológico, en los que fijando la noción de componente se busca definir la arquitectura lógica de ciertos tipos de sistemas y los pasos para poblarla con componentes para un sistema particular de uno de esos tipos. Por otra parte, se encuentran las tecnologías que proporcionan plataformas para el desarrollo y ejecución de componentes (o sistemas basados en componentes), como por ejemplo Java 2 Enterprise Edition (J2EE) [Sun01a]. En este mundo, la noción de componente se encuentra definida como una construcción de implementación. A pesar de que las plataformas existentes presentan puntos en común, cada una impone finalmente su enfoque particular, el cual va evolucionando con cada nueva versión de la tecnología. Podría considerarse que el primer mundo es independiente del segundo, en el sentido que busca expresar soluciones en términos de nociones abstractas (independientes de la tecnología) de componentes, enfoque que se denomina "basado en modelos". Sin embargo, dichos modelos deberán ser implementados utilizando las construcciones que una cierta tecnología brinde, por lo tanto se hace necesaria una conexión entre ambos mundos.

Es posible concebir una solución completamente en el mundo tecnológico, pero esto trae consigo algunos inconvenientes. Por ejemplo, la solución queda atada a una tecnología particular, o más aún, a una cierta versión de la tecnología. El razonamiento de la solución se podría ver distorsionado por el "ruido" introducido por la terminología y enfoque de la tecnología. Además, requeriría que el arquitecto de software no solo contara con un conocimiento adecuado de la tecnología a aplicar, sino que además fuera experto en la misma. El enfoque basado en modelos permite razonar la solución en forma abstracta e independiente de la tecnología y constituye un enfoque interesante para el manejo de la complejidad que representa el desarrollo de un sistema basado en componentes. Propuestas como la de [CD01, VP03] se enmarcan en este enfoque.

Existen en la actualidad esfuerzos por contemplar ambos mundos. El Object Management Group (OMG) con la iniciativa de Model Driven Architecture (MDA) [OMG01], busca separar la lógica de un negocio de la tecnología de la plataforma subyacente. De esa manera, aplicaciones independientes de la plataforma construidas según el enfoque de MDA pueden ser realizadas en diferentes plataformas propietarias. El concepto de modelo es ciertamente central en MDA. En particular se distinguen dos tipos de modelos: el PIM (platform independant model o modelo independiente de la plataforma), y el PSM (platform specific model o modelo específico de la plataforma). En MDA se transforma un PIM generado para un sistema en un cierto PSM. Luego, el PSM puede ser implementado directamente en la plataforma específica.

En este trabajo nos alineamos con el enfoque de MDA descrito. Partiendo de un modelo de un sistema realizado en forma independiente de la tecnología y según una metodología particular (es decir, un PIM generado aplicando una cierta metodología), buscamos definir una correspondencia con las construcciones de una tecnología particular. Eso quiere decir que el objetivo es brindar los elementos necesarios para la generación de un modelo específico de la plataforma (es decir la transformación a un PSM), y no la generación de código directamente desde el PIM. Concretamente los modelos PIM sobre los que nos concentraremos son los generados aplicando el enfoque de [CD01] el cual está a su vez basado en [DW98]. En él, se explican los pasos necesarios para especificar la arquitectura de componentes de un sistema de información, organizado en una arquitectura de capas, e independientemente de la tecnología. La plataforma para la cual se estudiará la transformación es J2EE. Esta transformación fue marginalmente estudiada en el propio [CD01], es nuestro objetivo dar un tratamiento tanto más profundo como más amplio a este punto.

El resto del documento se organiza de la siguiente forma. La sección 2 repasa las ideas fundamentales del enfoque metodológico de [CD01], y presenta un PIM resultado de su aplicación. En la sección 3 se presenta una correspondencia entre los tipos de componentes presentes en un PIM y las construcciones de J2EE, brindando una base para la generación de PSMs para dicha plataforma. La sección 4 presenta el PSM generado a partir de PIM de la sección 2 atendiendo lo expresado en la sección 3. La sección 5 concluye.

## 2. Arquitectura de Sistemas de Información Independientes de la Tecnología

Esta sección contiene un repaso del las principales ideas que se manejan en [CD01]. La arquitectura de un sistema de información se especifica en dos niveles de refinamiento. El primero, denominado *Arquitectura del Sistema* expresa el estilo de arquitectura a aplicar en el nivel más alto de abstracción. Como se dijera previamente, el estilo aplicado es el de capas. En la primera parte se definen concretamente las capas a utilizar, así como las principales responsabilidades de los componentes que residirán en cada una de ellas. En la segunda parte, en base a las responsabilidades mostradas, se estudian los elementos básicos de una especificación de *arquitectura lógica* de componentes, que constituyen los modelos independientes de la tecnología (PIM), así como un ejemplo de uno de ellos.

## 2.1 Arquitectura del Sistema

Los sistemas de información pueden modelarse según el estilo de arquitectura de *niveles de abstracción* o *capas* [SG96]. Este estilo se organiza jerárquicamente. Cada capa brinda servicios a la capa superior a ella y actúa como cliente para la capa inferior. Los servicios brindados en las capas altas corresponden a un nivel alto de abstracción, y los servicios brindados por capas bajas corresponden a un nivel bajo de abstracción. A continuación se presenta la organización de un sistema de información indicando las responsabilidades de los componentes que residen en cada capa. Esta organización está dada de forma que las capas están presentadas en orden decreciente de abstracción, y está basada en la propuesta por [CD01] habiéndosele agregado la capa de *IU de Diálogos del Usuario*:

- Capa de Interfaz de Usuario: Se encarga de la presentación al usuario. Contiene formularios, páginas Web, etc.
- Capa de IU de Diálogos del Usuario: Maneja la lógica de la IU y coordina la presentación al usuario.
- Capa de Diálogos del Usuario: Maneja la lógica de diálogos y mantiene el estado del diálogo.
- Capa de Servicios del Sistema: Exporta operaciones que resuelven requerimientos del sistema. Está organizada en subsistemas. Contiene además adaptadores a sistemas externos.
- Capa de Servicios del Negocio: Los componentes corresponden a tipos estables del negocio. Administra la información persistente del negocio.
- Capa de Infraestructura: Provee servicios básicos como seguridad, poder transaccional, caching de datos, etc. Comprende en general a frameworks, APIs, manejadores de bases de datos, etc.

## 2.2 Arquitectura Lógica

La arquitectura del sistema presentada antes indica el estilo general que seguirá la aplicación. El siguiente paso es determinar el estilo que cada capa seguirá internamente, así como identificar que elementos la poblarán.

La organización interna utilizada en cada capa varía. La interfaz de usuario, si es orientada al Web, seguirá el estilo *cliente-servidor*. La metodología de desarrollo puede condicionar el estilo de arquitectura a elegir, ya que mediante la aplicación de la metodología cada capa será poblada con elementos que residirán en ella. El ejemplo de la Figura 1, tratado en [PV03], fue desarrollado aplicando el enfoque de [CD01, VP03]. En consecuencia, el estilo de arquitectura utilizado en las capas Diálogos de Usuario, Servicios de Sistema y Servicios de Negocio es *componentes*.

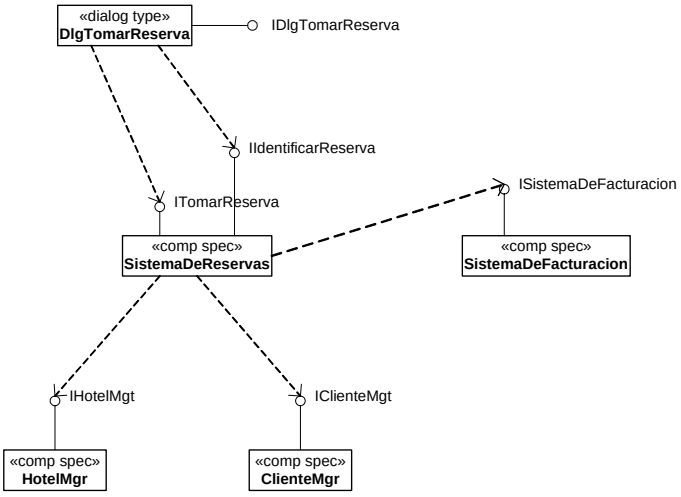



Figura 1 – Arquitectura lógica independiente de la tecnología

El estereotipo «comp spec» denota a una especificación de componente. Cada especificación ofrece una o más interfaces, las cuales son estereotipadas con «interface type». Una interface type tiene asociado un modelo de información sobre el cual se expresan los contratos de software para cada una de sus operaciones. Los diálogos de usuario no son considerados en [CD01]. Un diálogo de usuario, estereotipado con «dialog type», encapsula la lógica de un caso de uso. Por no tener un modelo de información asociado, la interfaz provista no necesita ser especificada mediante un interface type, una simple interfaz es suficiente para lo que se necesita especificar de un diálogo.

## 3. Correspondencia a la Plataforma J2EE

En esta sección se discuten los detalles de la correspondencia entre los elementos de cada una de las capas de la arquitectura del sistema y las construcciones de la plataforma J2EE. Con esta correspondencia es posible transformar un modelo tipo PIM, como el mostrado en la arquitectura lógica, en uno específico para J2EE tipo PSM. Al aplicarse EJB [Sun01b] a las capas que utilizan el estilo de componentes, para su modelado se aplicó el profile de UML para EJB [Gre01]. Se atacarán una a una las capas indiadas en la arquitectura del sistema, basando los ejemplos en la arquitectura lógica ya presentada.

### 3.1 Interfaz de Usuario

Esta capa no fue incluida en la arquitectura lógica. Con el avance de las tecnologías de Internet, cada vez más empresas orientan sus aplicaciones al web, ya que cuentan con una Intranet. Teniendo esto en cuenta, realizaremos el mapeo de forma de utilizar tecnología web.

Esta capa se organiza generalmente con el estilo cliente-servidor. En el cliente corre un browser. Este despliega al usuario final páginas web basadas en HTML y HTML Forms. Mediante estos últimos se recopila información que será comunicada al servidor.

A nivel del servidor se cuenta con un servidor Web capaz de generar páginas web en forma dinámica. La tecnología Java Server Pages (JSP) [Sun01c] es el mecanismo usado para este fin. Las páginas JSP se construyen a partir de un lenguaje de tags y permite definir páginas dinámicas a través de templates. Estos templates incluyen tags HTML para la parte estática y tags especiales JSP para las partes en donde se debe incluir contenido dinámico. Para una interacción completa Actor-Sistema (i.e. un caso de uso) es necesario, en general, más de una página web. Se utilizará una página JSP por cada etapa de la interacción. Los forms serán dirigidos (mediante el correcto seteo del atributo `action`) a una URL atendida por un *servlet* [Sun01d] ubicado en la capa inferior. A su vez, este servlet es el responsable de proporcionarle a la página la información necesaria para generar el contenido dinámico. Para esto se utiliza el EJB pattern Data Transfer Object (DTO)[1] [Mar02]. Este define que la información debe estar empaquetada en clases serializables, para lo cual utilizaremos JavaBeans [Sun97]. Este en general será el mecanismo utilizado para el intercambio de información entre todas las capas del sistema.

### 3.2 UI de Diálogos del Usuario

Esta capa aplica una variante del GRASP [Lar01] facade controller. Todas las páginas dirigen sus respuestas a este controlador. Se utiliza un servlet para este fin[2]; el cual es el encargado de recibir la información del usuario (recibida mediante la solicitud http) e invocar al diálogo de usuario (use-case controller) correspondiente al caso de uso. El servlet a través de su objeto sesión tiene asociado un diálogo de usuario al cual redirigirá todas las solicitudes. En cada solicitud envía la información recibida (encapsulándola en un JavaBean), obtiene el resultado, y responde al usuario con una nueva página JSP, la cual se construye con los datos recibidos del diálogo.

Notar que de la forma en que se define este mapeo, es una aplicación del pattern Model-View-Controller [BMR+96] en donde las páginas JSP cumplen el rol de “View”, el servlet cumple el rol de “Controller” y las capas que se presentan a continuación el rol de “Model”. Otra alternativa al modelado de estas dos capas es utilizar el framework Struts [HDF+03], lo cual se propone como trabajo futuro.

### 3.3 Diálogos de Usuario

El diálogo de usuario encapsula la lógica del caso de uso. Así, implementa una máquina de estados (derivada a partir del documento de casos de uso expandido), la cual es utilizada para saber qué servicios y en qué orden requerir a la capa inferior, así como para determinar la próxima página a visualizar. El servlet definido para la capa anterior no implementa esta máquina de estados. No vamos a exigirle que conozca cual es la operación en el diálogo que corresponde invocar a partir de una solicitud. Para ello vamos a aplicar el design pattern Command. Definimos una interfaz `IDialog` cuya única operación es `execute(in datain:Data):Result`.

---

[1] O una de sus variantes.

[2] Los J2EE BluePrints recomiendan que los servlets sean utilizados en actividades con una importante componente de programación. Esto lo deja como el candidato ideal para cumplir el rol de controlador de procesos.

`Data` es la superclase de todos los datos que son intercambiados entre cada diálogo y la capa superior, y se define como JavaBean. Así mismo, `Result` es otro JavaBean que contiene una instancia de `Data`, que representa a los datos resultantes, y además un identificador del estado en que quedó el diálogo luego de terminar la ejecución de `execute`. El servlet (en la capa superior) utiliza el identificador del estado para elegir la siguiente página JSP, a la cual le provee la instancia de `Data` incluida en la instancia de `Result` recibida. Notar que este mapping puede definirse mediante un archivo de configuración (basado en XML por ejemplo), de forma de no necesitar cambiar el código del servlet si sufre cambios las páginas JSP o los diálogos de usuario. Un diálogo de usuario es implementado mediante un *stateful session bean (SSB)*, aplicando el design pattern State. El SSB es un tipo de EJB que permite mantener el estado conversacional con el cliente (en nuestro caso la máquina de estados) a lo largo del proceso.

En resumen, un escenario de acción entre las tres capas especificadas hasta el momento es el siguiente. El usuario llena los datos del formulario presentado en su browser. Al hacer submit la información viaja al servidor web y es recibida por el servlet. El mismo invoca la operación `execute` del diálogo con el cual fue instanciado pasándole el JavaBean (instancia de `Data`) que encapsula la información recibida del submit realizado por el usuario. El diálogo, en función de su estado y de la información recibida, solicita servicios a la capa Servicios del Sistema, obtiene un resultado, decide su siguiente estado, y retorna la instancia de `Result` que incluye la instancia de `Data` correspondiente al resultado y el identificador del nuevo estado. Así, el servlet dirige al cliente web a la página JSP correspondiente al estado recibido, construyéndola con la información recibida en la instancia de `Data`.

Es importante distinguir que granularidad de transaccionalidad es necesaria. Utilizando *container-managed transaction* (transaccionalidad manejada por el contenedor), solamente es posible demarcar transacciones a nivel de operaciones. Esto significa que si utilizamos transaccionalidad manejada por el contenedor para la interfaz del dialogo de usuario una transacción solo puede estar relacionada a un método de esta. En cambio, lo que se necesita es que el caso de uso entero este enmarcado dentro de la misma transacción. Para ello se debe utilizar *bean-managed transaction* (transaccionalidad manejada por el bean). En este caso, cuando se recibe la primera invocación a la operación `execute`, se crea una nueva transacción. Como se está utilizando un stateful session bean, la transacción será preservada entre una invocación y otra. La máquina de estados implementada en el bean podrá indicar cuando debe realizarse el commit, o hacer rollback en caso de que el caso de uso falle.

Se presenta a continuación la vista interna del stateful session bean `DlgTomarReserva`.

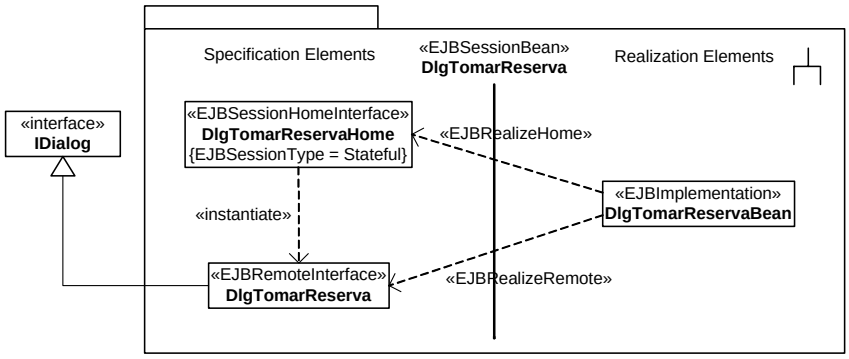



Figura 2 – Vista interna del session bean `DlgTomarReserva`

Notar que la Component Interface[3] del diálogo de usuario extiende la interfaz IDialog mencionada anteriormente.

### 3.4 Servicios de Sistema

Un proceso de negocios da lugar a un conjunto de casos de uso. Siguiendo la metodología, se crea una interfaz de sistema por cada caso de uso detectado. Estos casos de uso, originados a partir del mismo proceso, presentan una alta cohesión. Así, se crea un único componente de sistema que realiza todas las interfaces originadas por dicho proceso.

---

[3] El modelo de componentes EJB define que un componente debe ser accedido a través de su Component Interface. Esta representa la especificación que el componente realiza. También define dos variantes, la Remote Interface que permite que el componente sea accedido remotamente y la Local Interface que permite que el componente sea accedido localmente.

En el modelo de componentes EJB cada componente implementa una y solo una Component Interface. Para lograr que que un componente implemente más de una interfaz, es posible que la Component Interface sea sub-interfaz de todas las interfaces provistas por el componente. Determinar si la Component Interface es Remota o Local no es un tema menor[4]. En el común de los casos, en donde la capa de "Diálogos de Usuario" está ubicada en el mismo servidor de aplicaciones que la de "Servicios del Sistema", lo más performante es hacer que la Component Interface sea Local. Si por otro lado, el caso es que interesa distribuir estas capas sería necesario tener interfaces Remotas. Este mismo criterio se utilizará para determinar si las interfaces de las capas inferiores deben ser remotas o locales.

Para el caso de estudio se considerará que las capas de "Dialogo de Usuario", "Servicios de Sistema" y "Servicios de Negocio" están ubicadas en el mismo servidor de aplicaciones, por lo cual se utilizarán interfaces Locales.

El componente creado actúa de fachada del sistema para el proceso, agrupando todas las operaciones involucradas en él. Los componentes en esta capa cumplen el rol de *Facade Controller* siguiendo GRASP nuevamente.

La información de la sesión referente al diálogo entre el Actor y el Sistema es mantenida en la capa superior; los componentes a este nivel no necesitan mantener estado. Por ello se utilizará *stateless session beans*. Notar que la capa de servicios del sistema sigue los lineamientos del EJB pattern Session Facade [Mar02].

Es importante notar que generalmente todas las operaciones a este nivel necesitan estar enmarcadas en una transacción. Esto se indica en forma declarativa marcando cada una de las operaciones con el atributo `TX_REQUIRED`.

Se presenta a continuación la vista interna del stateless session bean `SistemaDeReservas`.

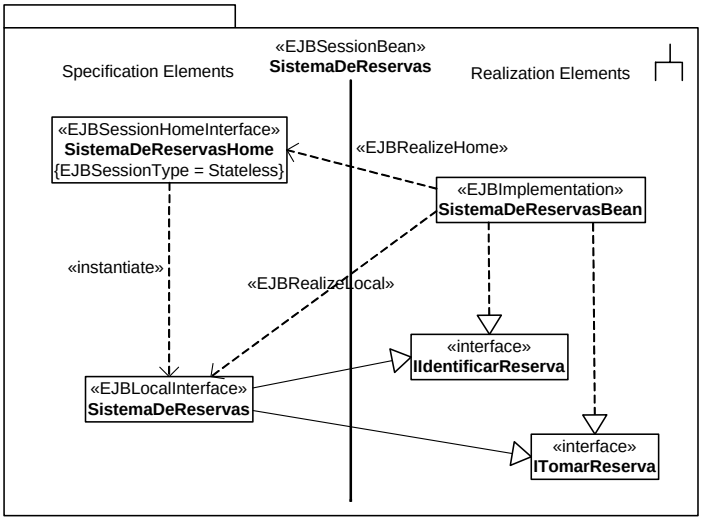



Figura 3 – Vista interna del session bean SistemaDeReservas

### 3.5 Servicios de Negocio

Los componentes en esta capa fueron detectados a partir de un Modelo de Tipos del Negocio. La metodología indica que se debe crear una interfaz de negocio responsable de manejar todas las instancias de cada tipo principal (conocido como *core type*) del modelo (e.g. `IClienteMgr`). Para acceder y procesar dichas instancias debe pasársele al *manager* los identificadores (e.g. `ClienteId`) de las instancias sobre las cuales trabajar. Por ejemplo, una operación `getDetallesCliente()` de `IClienteMgr` recibe el identificador del cliente y retorna los detalles del mismo.

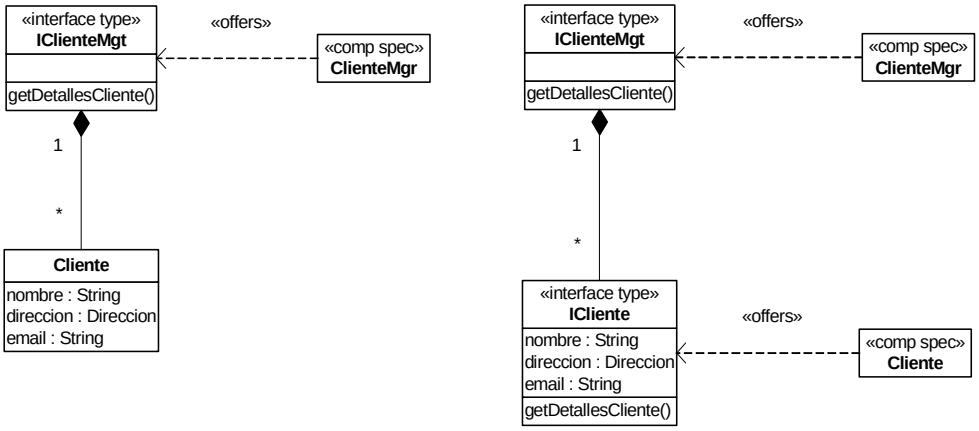
Al implementar un manager en EJB se utilizará también un stateless session bean. Cada componente que necesite servicios del manager deberá solicitar una instancia del mismo. Este enfoque es simple, aplicable en muchos casos y da soporte a las interfaces tal cual fueron especificadas.

---

[4] Notar que esto no sería un problema si la especificación EJB permitiera manejar transparentemente el hecho que el componente sea accedido remota o localmente. Esto claramente es una limitante de la especificación actual.

Una vez decidido que es un session bean, debe definirse en que forma el manager va a manipular la información que administra y se encuentra en una o varias fuentes de datos (*data sources*).

Un primer enfoque es que los session bean utilicen Java DataBase Connectivity (JDBC) [Sun01e] para manipular la información. Una clara desventaja de este enfoque es que el desarrollador debe implementar explícitamente la persistencia de su aplicación. Lo cual lleva a que no se logre una clara separación en lo que es la manipulación de los tipos de negocio y la persistencia de los mismos. Aparte puede ocasionar problemas conocidos como SQL-Spagueti. Para minimizar estos problemas se puede aplicar el J2EE pattern DataAccessObject [ACM01] de forma de encapsular el acceso a las fuentes de datos. Otra alternativa interesante para manejar la persistencia es utilizar Java Data Object (JDO) [Sun03, Mar02] como mecanismo de persistencia. El mismo provee un mapeo automático de un modelo de objetos en Java al modelo relacional. Aparte soporta características interesantes como transacciones y seguridad. Si bien esta tecnología no es parte de la plataforma J2EE puede ser integrada fácilmente, como se explica en [Mar02]. Profundizar en JDO escapa al alcance del presente artículo. Ambos enfoques tienen en común que desde el punto de vista de los componentes, el session bean que actúa como manager es el responsable de manejar la persistencia de su información. La parte (a) de la siguiente figura esboza este enfoque.




(a) Persistencia manejada por el Manager (b) Persistencia manejada con Entity Beans

Figura 4 – Dos enfoques para implementar los servicios de negocio

Una alternativa a este enfoque es continuar la aplicación del modelo de componentes bajo el nivel del manager, considerando a las instancias de `Cliente` como instancias de componente (*component object*) propiamente. Con este enfoque se simplifica a su vez la especificación de alguna de las funcionalidades del manager ya que habrá menos necesidad de des-referenciar instancias. Las operaciones específicas a una instancia serán definidas en una interfaz que realice al interface type del Cliente, a saber `ICliente`. Podemos utilizar aquí un *entity bean* para implementar la interfaz `ICliente`, donde cada instancia del entity bean representa a un Cliente. Esto nos permite utilizar *container-managed persistente* para la recuperación y almacenamiento de las estructuras de un cliente. A su vez, a partir de EJB 2.0 [Sun01b], podemos utilizar EJB Query Language (EJB-QL) para consultar la información independizándonos así del lenguaje de consulta de la fuente de datos. En la parte (b) de la Figura 4 – se esboza este enfoque.

Este enfoque permite aprovechar las bondades del contenedor de EJB a nivel de las entidades del negocio (seguridad, transacciones, pooling, cache,, fail-over, clustering[5], entre otras). En particular promueve la portabilidad de los componentes, la cual se vería afectada en el caso de utilizar JDO (por no ser una tecnología J2EE) o JDBC (si cambiamos el motor de base de datos subyacente).Pero por otro lado, para determinadas operaciones puede ser muy ineficiente, por ejemplo, las operaciones que afectan varios entity beans. Está fuera del alcance de este trabajo una completa comparación de ambos enfoques. Por más información ver [Mar02].

[5] Cabe notar que Fail-Over y Clustering no son funcionalidades estándares de un contenedor, pero la mayoría las implementan.

El enfoque que se utilizará para modelar el resto del caso de estudio será el basado en entity beans. La vista interna del session bean `ClienteMgr` se presenta a continuación.

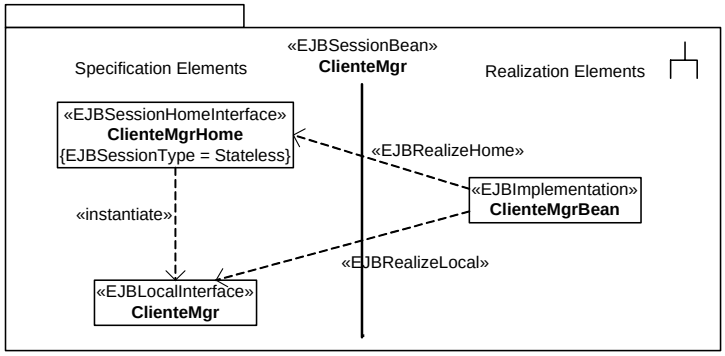



Figura 5 – Vista interna del session bean `ClienteMgr`

La vista interna del entity bean `Cliente` se presenta a continuación.

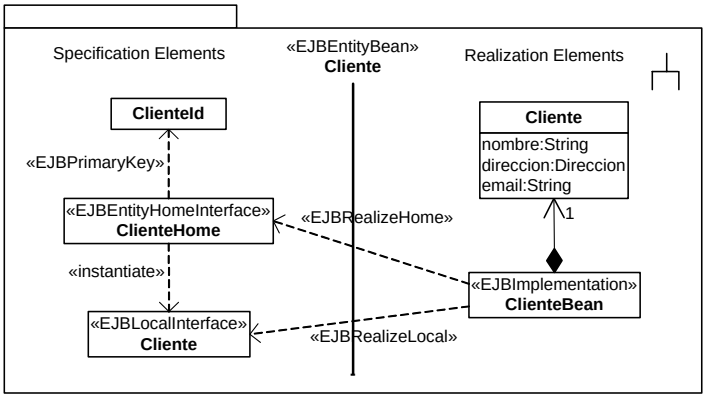



Figura 6 – Vista interna del entity bean `Cliente`

Cabe notar que tanto los entity beans como los session beans de esta capa exportan sus servicios mediante interfaces locales. En el caso particular de los entity beans, se recomienda que siempre sean accedidos a través de sus interfaces locales, lo cual implica que la capa de servicios de negocio estén en el mismo contenedor.

### 3.6 Integración de Sistemas Existentes

La integración con sistemas existentes puede realizarse utilizando las APIs provistas por el sistema existente o software para la integración de aplicaciones (EAI – Enterprise Application Integration).

El caso de estudio en consideración utiliza el sistema existente de Facturación en uso en la empresa. Será necesario desarrollar un adaptador (aplicación del design pattern Adapter) que provea un puente entre el entorno de componentes y el sistema externo. Siguiendo con el mapeo a la tecnología EJB, se utilizará J2EE Connector Architecture (JCA) [Sun02] para acceder vía Common Client Interface (CCI) a las funcionalidades provistas por el sistema externo. Utilizando JCA, y asumiendo que se dispone del adaptador XA-Resource para el sistema existente, puede incluírselo en las transacciones.

Se incluirá entonces un session bean que dará soporte a la interfaz necesaria del sistema existente. Este bean redireccionará (probablemente necesite adaptar datos que entran y salen) las solicitudes al sistema externo. De esta forma centralizamos en un único componente el uso de JCA para acceder al sistema externo, preservando así las cualidades deseadas de un sistema basado en componentes.

La vista interna del session bean `SistemaDeFacturacion` se presenta a continuación.

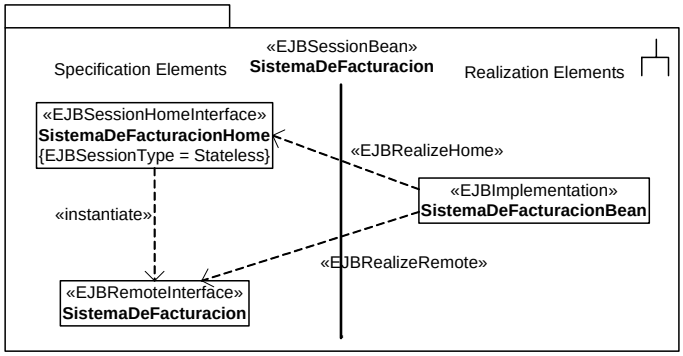



Figura 7 – Vista interna del session bean `SistemaDeFacturacion`

## 4. Ejemplo de Modelo Específico para la Plataforma J2EE

En esta sección se presenta el modelo tipo PSM correspondiente al modelo tipo PIM presentado en la Figura 1, el cual se muestra en la Figura 9. Esta vista unifica las distintas correspondencias estudiadas en la sección 3, refiriéndose solamente a aquellas capas pobladas con componentes, i.e. Diálogos, Sistema y Negocio.

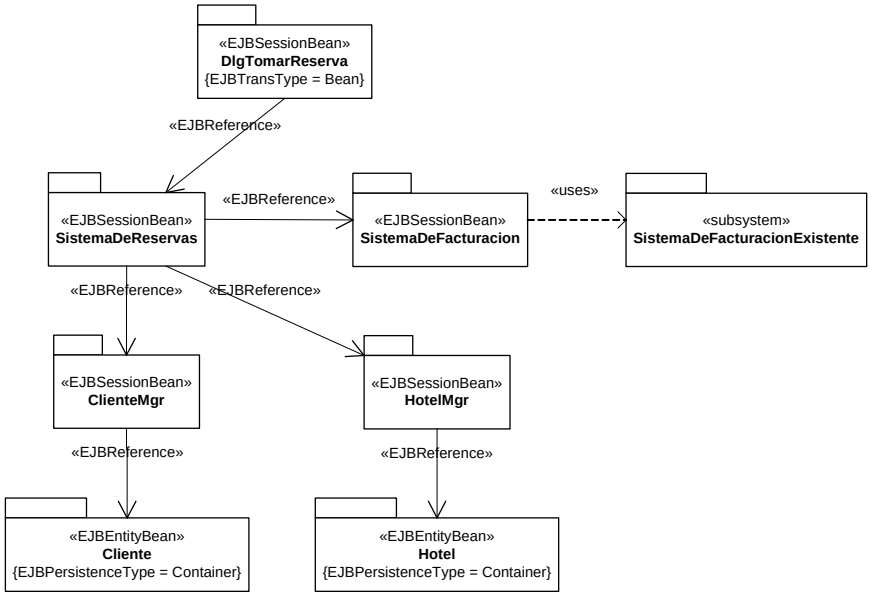



Figura 8 – Arquitectura lógica específica para J2EE

En el caso de estudio abordado en este trabajo, puede observarse que los modelos de cada figura presentan una estructura similar. El hecho de que en la Figura 9 se cuente con dos beans correspondientes a un componente de negocio (e.g. la especificación de componente ClienteMgr se corresponde con el session bean ClienteMgr y el entity bean Cliente) refiere a haber optado por utilizar entity beans; de la aplicación de un enfoque basado en JDBC o JDO no se obtendría el mismo modelo.

## 5. Conclusiones

La arquitectura de los sistemas de información suele organizarse según el estilo de capas. La cantidad de capas y la responsabilidad de cada una dependen del enfoque metodológico aplicado para el desarrollo del sistema. En [CD01] se define una arquitectura para dichos sistemas y se presenta una propuesta orientada a la especificación independiente de la tecnología de implementación de componentes como los habitantes de algunas de las capas. Una arquitectura de

especificación para un sistema particular corresponde a un PIM en el enfoque de MDA. En este artículo se refinó la arquitectura propuesta para los sistemas de información añadiendo una nueva capa, la capa de *IU de Diálogos del Usuario*, y se definió una correspondencia entre los elementos de cada una de las capas de la arquitectura del sistema y las construcciones de la plataforma J2EE. Esta correspondencia permite realizar una transformación a una arquitectura de especificación. El resultado de dicha transformación se expresa en términos específicos de la plataforma, y corresponde a un PSM en el enfoque de MDA. La separación entre PIMs y PSMs, y la definición de transformaciones de unos a otros constituyen la esencia del enfoque de MDA. Este trabajo presenta un estudio que hace posible una de esas transformaciones.

Finalmente, resulta de interés completar el ejemplo utilizado en el artículo, generando la arquitectura lógica específica para J2EE del caso de estudio de [PV03], con el objetivo de realizar una implementación completa que permita refinar la correspondencia presentada. También resulta de interés modelar las dos capas más superiores utilizando el framework Struts, así como el estudio de correspondencias análogas a otras tecnologías, como ser la plataforma .NET.

## Referencias


[ACM01] D. Alur, J. Crupi, D. Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice-Hall, 2001.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, 1996.

[CD01] J. Cheesman, J. Daniels. *UML Components: A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2001.

[DW98] D.F. D'Souza, A.C. Wills. *Objects, Components and Framekorks with UML: The Catalysis Approach*. Addison-Wesley, 1998.

[GHJ+95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Prentice-Hall, 1995.

[Gre01] J. Greenfield. *UML Profile for EJB*. Rational Software Corporation, Public Draft, 2001.

[HDF+03] Ted Husted, Cedric Dumoulin, George Franciscus, David Winterfeldt. *Struts in Action*. Manning, 2003.

[Lar01] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process (2nd Edition)*. Prentice-Hall, 2001.

[Mar02] Floyd Marinescu. EJB Design Patterns. Advanced Patterns, Processes and Idioms. John Wiley&Sons, 2002.

[OMG01] OMG. *Overview and Guide to OMG's architecture, MDA Guide Versión 1.0*. Object Management Group, 2001.

[PV03] D. Perovich, A. Vignaga. *SAD del Subsistema de Reservas del Sistema de Gestión Hotelera*. Reporte Técnico RT03-15, InCo Pedeciba, Montevideo, Uruguay, 2003.

[SG96] M. Shaw, D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[Sun97] Sun Microsystems. *JavaBeans Specification*. Versión 1.0.1, 1997.

[Sun01a] Sun Microsystems. *Java 2 Enterprise Edition Platform Specification*. Versión 1.3, 2001.

[Sun01b] Sun Microsystems. *Enterprise JavaBeans Specification*. Versión 2.0, 2002.

[Sun01c] Sun Microsystems. *JavaServer Pages Specification*. Versión 1.2, 2001.

[Sun01d] Sun Microsystems. *Java Servlet Specification*. Versión 2.3, 2001.

[Sun01e] Sun Microsystems. *Java DataBase Connectivity Specification*. Versión 3.0, 2001.

[Sun02] Sun Microsystems. *J2EE Connector Architecture*. Versión 1.5, 2002.

[Sun03] Sun Micrsystems. *Java Data Objects Specification*. Versión 1.0, 2003.

[VP03] A. Vignaga, D. Perovich. *Enfoque Metodológico para el Desarrollo Basado en Componentes*. Reporte Técnico RT03-14, InCo Pedeciba, Montevideo, Uruguay, 2003.